Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Raoul Udd

Adopting Continuous Delivery:
A Case Study

Master's Thesis
Espoo, March 21, 2016

Supervisor: Professor Casper Lassenius
Advisors: Juha Itkonen D.Sc. (Tech.)
Eero Laukkanen M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Raoul Udd

Title:
Adopting Continuous Delivery: A Case Study

Date:	March 21, 2016	**Pages:**	76
Major:	Software Engineering and Business	**Code:**	T-76
Supervisor:	Professor Casper Lassenius		
Advisors:	Juha Itkonen D.Sc. (Tech.) Eero Laukkanen M.Sc. (Tech.)		

Continuous delivery (CD) is a practice that builds upon the concept of continuous integration. When developing software with CD, every change that passes through the deployment pipeline results in a fully working product that can be deployed without effort. This practice has the potential to accelerate value delivery, improve the software quality and increase developer productivity.

The goal of this thesis is to investigate the adoption of CD and evaluate the results of the adoption in a single case organization. This is done through a single case study, primarily on the basis of qualitative data from interviews but also utilizing quantitative data from tools used in the development environment.

The study shows that the multi-year transition included adoption of many of the typical methods and tools reported in existing research. This includes construction of a deployment pipeline, automation of tests and employment of environment independent builds. Increased communication and collaboration between developers and stakeholders was a major enabler of the adoption, but can also be seen as a beneficial outcome. Other reported benefits of the transition was increased productivity, improved product quality, improved developer morale as well as infrastructural and organizational agnosticism. Exploratory analysis of ticket system metadata did not reveal any definite quantitative results of the adoption, but showed that metrics from different systems can be used to evaluate and reason about the progress of CD adoption.

In the case studied, CD was achieved despite the obstacles introduced by the heavily coupled systems under development and legacy code base. Positive outcomes of the transition were observed by both the developing organization and customer.

Keywords: Continuous Delivery, Continuous Integration, Single Case Study, Transformation, Qualitative Analysis

Language: English

Aalto-universitetet
Högskolan för teknikvetenskaper
Examensprogram för datateknik

SAMMANDRAG AV
DIPLOMARBETET

Utfört av:	Raoul Udd		
Arbetets namn:	Att ta i bruk kontinuerling leverans: En fallstudie		
Datum:	21 mars 2016	**Sidantal:**	76
Huvudämne:	Programvaruproduktion och affärsverksamhet	**Kod:**	T-76
Övervakare:	Professor Casper Lassenius		
Handledare:	TkD Juha Itkonen DI Eero Laukkanen		

Kontinuerlig leverans (KL) är en praxis som bygger vidare på kontinuerlig integration av programvara. När man utövar KL vid programvaruutveckling så resulterar varje ändring av källkoden som tar sig igenom alla leveranspipelinens steg i en fullt fungerande produkt som kan sättas i drift utan möda. Denna praxis kan potentiellt accelerera leveransen av värde, höja programvarans kvalitet och öka på utvecklarnas produktivitet.

Målet med detta arbete är att undersöka ibruktagandet av KL samt att utvärdera resultaten av ibruktagandet i en organisation. Detta görs genom en enfallstudie, med kvalitativ data från intervjuer som primärkälla samt kvantitativ metadata från verktyg som används i utvecklingsmiljön.

Studien visar att den fleråriga transformationen inkluderade ibruktagandet av många av de typiska metoder och verktyg som rapporterats i existerande forskning. Detta innebar t.ex. byggandet av en leveranspipeline, automatisering av tester samt övergången till miljöoberoende byggen. Ökad kommunikation och samarbete mellan utvecklare och intressenter var en viktig möjliggörande faktor för övergången, och kan också se som ett gynnsamt resultat. Andra fördelar med KL i detta fall är den ökade produktiviteten, förbättrade produktkvaliteten, höjd arbetsmoral samt organisatorisk och infrastrukturell agnosticism. Explorativ analys av metadata från ärendehanteringssystemet avslöjade inte några tydliga kvantitativa resultat av övergången till KL, men visade att mätare från olika system kan användas för att utvärdera och resonera om ibruktagandet.

I fallet som studerades uppnåddes KL trots de hinder som utgjordes av de kraftigt ihopkoppade systemen under utveckling och den föråldrade källkoden. De positiva resultaten av övergången observerades såväl av den utvecklande organisationen som av kunden.

Nyckelord:	Kontinuerlig Leverans, Kontinuerlig Integration, Enfallstudie, Transformation, Kvalitativ Analys
Språk:	Engelska

Acknowledgements

I extend the deepest of gratitude towards my colleagues in the Software Process Research Group at Aalto University. I want to thank my professor Casper Lassenius for guiding and supporting me throughout the process, as well as Juha Itkonen and Eero Laukkanen for selflessly helping me and partaking in the research out of their own passion for the topic. A great big thank you goes out to the employees of Solita, both Timo Lehtonen for enthusiastically driving the study forward and Janne Rintanen along with the developers for providing valuable insight into the case. Lastly, we are grateful for the input from the employees of Tekes, who welcomed us with open arms.

Espoo, March 21, 2016

Raoul Udd

Abbreviations and Acronyms

CD	Continuous Delivery or Continuous Deployment
CI	Continuous Integration
VCS	Version Control System
DB	Database
Developing organization	A group of people developing software as a way of producing value, often as part of a company and as one or several teams.

Contents

Chapter 1

Introduction

Software development is — by nature — a complex task, performed in rapidly changing contexts and in an environment with high amounts of competition and uncertainty. Since its inception, practicians and academics alike have sought to improve the practices and processes by which software is created. Agile software development is arguably the most significant paradigm to arise during the last decades, impacting the ways by which value is delivered to customers and end users. As agile methodologies are now widely adopted, both developing organizations and customers are beginning to recognize the value of software being delivered quickly and with high quality.

Continuous integration (CI) has gained traction as a means of tackling issues with visibility of development status and feedback on code quality. This is done through automation of building, testing and integrating each commit using an integration pipeline. By adopting the discipline properly, bugs are dealt with as they are conceived, and frustrating, unpredictable code integrations just before a planned release can be avoided. Thus the practice intends to allow for quicker delivery of value to the customer, one of the main principles of agile. One of the main goals of continuous integration is to keep the code base in such a state that it is quickly deployable. When code is integrated often and there is no vast backlog of bugs, time to deployment is heavily reduced. [Fowler, 2006]

Continuous delivery (CD) builds upon the concept of continuous integration by extending the integration pipeline into a deployment pipeline. When successfully performing continuous delivery, the software under development can be released into production whenever is needed [Fowler, 2013]. Thus, not only is the integration of the system automated, but also the staging required to reliably release quality software into a production environment, and the deployment itself.

This aim of this thesis is to examine the adoption and measurement of CD.

This is performed through studying a customer project in a case organization, where CD has been pursued over the course of several years. The research intends to give an understanding of the changes needed to adopt CD and the value of making those changes through answering the following research questions:

RQ1 How has the case organization been adopting CD?

RQ2 What have the outcomes of the adoption of CD been?

RQ3 How can we measure the success of the adoption of CD?

These questions are answered by the means of a descriptive single case study [Yin, 1994]. Data used in the study includes quantitative data collected from systems and tools in use in the case organization, along with qualitative data gathered through interviews with key employees in the case organization and the customer.

1.1 Structure of the Thesis

This thesis is divided into nine subsequent chapters. In the following chapter, we will review the existing literature on the topic of CD and present the proposed benefits as well as typical characteristics of a successful CD implementation. In the third chapter, the context of the case under study is described. The fourth chapter takes a closer look on the method used in the study. After this, the research questions will be answered through analysis of collected data in chapters five, six and seven. Lastly, in chapters eight and nine, we will discuss the findings and draw conclusions from the research.

Chapter 2

Background

When producing software to cover the needs of its intended users, long periods between releases constitute a large risk. Getting useful feedback on whether or not the software under development satisfies the actual requirements of a large number of users is virtually impossible before they can try it out themselves. Thus, reducing the time it takes for a new feature to make it into production and shortening the feedback loop decreases the discrepancy between what developers and users understand as value. No developing organization or customer should want to spend large amounts of time, effort and money to develop the wrong product.

When organizations perform continuous delivery (CD), the code they produce is built and treated in a way where the resulting software can be released at all times [Fowler, 2013]. To be able to achieve this, the manual steps that have traditionally been performed before a planned release must be automated and performed continuously. All the code that is produced must be checked in to a version control system and the whole software integrated and built into executables, which are then automatically tested. This series of actions is what practitioners in the field of software engineering call continuous integration (CI) [Fowler, 2006]. In order to make sure that the software is releasable, however, the executables need to continue their path by installation into more and more production-like environments, after the CI pipeline. When the entirety of this pipeline is automized, and every successful code commit ends up in an actual production environment, we use the term continuous deployment. Continuous delivery is different from continuous deployment in that some manual action is required to actually deploy the release. Thus, the decision can be made whether or not to deploy a particular product increment. [Fowler, 2013]

This chapter is a review of CD from the perspective of previous literature. For the purpose of history and context, the practice of CI is presented in

section 2.1. Building upon this, and as background for the research questions, the characteristics of successful CD are presented in section 2.2 and the proposed and documented benefits of CD in section 2.3.

2.1 Continuous integration

The first written mention of continuous integration was made by Kent Beck as one of the many practices that are part of Extreme Programming [Beck, 2000]. It has since become a core practice of agile software engineering. Martin Fowler, a prominent figure in bringing CI to the attention of practicians, describes CI as a software development practice where teams integrate their code early and often in order to enable quick delivery of software and reduce integration problems [Fowler, 2006]. Before the wide adoption of CI, and in many projects still, code could not be considered working before it was proven to work through a tedious process of integration and testing, usually performed when development was considered "done" and a release was scheduled. With CI, working software is a default, and any broken integration of new code redirects the team's focus towards fixing the issue instead of developing new features. [Humble and Farley, 2010]

This practice requires developers to check in their work to a version control system every time they make a cohesive increment to the software. After this, the CI pipeline integrates the change with the rest of the code and runs any available tests. This sequence of actions is triggered automatically as a commit is made. An example of a CI pipeline can be seen in Figure 2.1. If something fails at any stage of the pipeline, the build is considered broken. In any case, the developer is informed about the result of the build, typically by email. Until the developer receives a notification that the integration was successful, he or she is not done with the commit. [Fowler, 2006]

CI intends to tackle several challenges with software development. The integration step before a release used to be one of the most critical parts of a projects lifecycle and on top of that, one of the most unpredictable ones [Fowler, 2006]. CI should remove this step from the process entirely, as software is integrated and tested continuously. Furthermore, CI enables a team to deal with bugs with increased efficiency and reduced risk [Fowler, 2006]. As bugs accumulate in the software, undiscovered, they get increasingly hard to remove due to interdependencies. Also, a bug discovered later in development or after release is much more expensive to fix than one that has just been created and thus can be easily pinpointed. CI also serves as a communication tool, allowing stakeholders to get an overview of the development status at any given time [Humble and Farley, 2010].

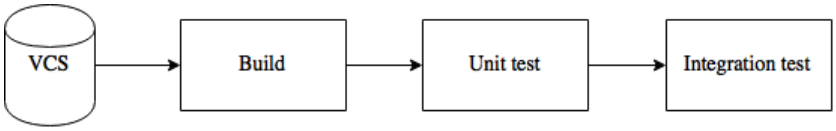


Figure 2.1: An example of a typical CI pipeline

The practice of continuous integration constitutes the foundation of continuous delivery. In many ways, CD is just the natural evolution of CI closer to the end customer.

2.2 Characteristics of continuous delivery

In this section, we will review what an ideal CD process should look like based on previous research. This will serve as a basis for answering RQ 3 on how well the case organization has achieved CD. A successful adoption of CD relies on organization-wide change [Humble and Farley, 2010]. As such, this chapter is divided into three subsections. First, the deployment pipeline is discussed, after which some tool options for CD implementation are reviewed. Then, the practices and principles that teams and organizations should conform to are presented. A summary of the characteristics can be found in table 2.1.

2.2.1 The deployment pipeline

Just like with CI, the pipeline lies at the heart of CD. The CD pipeline differs from the CI pipeline in that it not only integrates and tests the code in a development environment, but moves the executables forward into increasingly production-like environments, often including automated acceptance tests as well. After a commit has passed through all stages of the pipeline successfully, the software is proven to be potentially deployable. The actual deployment should be automated too, so that the developing organization can release the current version of the software at a moments notice at the request of a stakeholder. [Humble and Farley, 2010]

The typical stages involved in a CD pipeline are described below, and an example of a CD pipeline is depicted in section 2.5. Note, however, that there is no single correct way of implementing this pipeline, and the stages included may vary depending on the context and needs of the case at hand. The important rule of thumb is to make sure that the output of the pipeline is a releasable piece of software.

	[Humble and Farley, 2010]	[Fowler, 2013]	[Neely and Stolt, 2013]	[Chen, 2015]	[Leppänen et al., 2015]
CD pipeline	x	x	x	x	x
Small code changes per release	x	x	x		x
Fast automated tests	x	x		x	
Deployable software over new features	x	x			
Organizational support and collaboration	x	x	x	x	x
Build binaries once, deploy the same way	x				x
Deployment by the push of a button	x	x	x		x
Information radiators and metrics	x	x	x		

Table 2.1: A summary of the characteristics of CD in existing literature.

The commit stage. The first stage of the pipeline can be considered a summary of a basic CI pipeline. The commit stage is triggered when a developer commits new code to the source code repository. From here, the code is compiled. If compilation is successful, a test suite is run which usually consist of unit tests at this stage. The binaries created here should be the same binaries that will eventually flow through the entire pipeline. It is also common to perform code analysis in the commit stage, to check the health of the code according to metrics such as duplicated code amount, test coverage and code style. If any of these metrics don't reach a set threshold, the pipeline should be halted, as sufficient quality may not be assured. Lastly, any artifacts (test databases etc.) needed for the later stages are generated. If any of these substages fail, the execution of the pipeline is immediately aborted and the committing developer is informed about the failure. [Humble and Farley, 2010]

The acceptance test stage. While the commit stage has proven that the technical aspects of the software are in order, the acceptance test stage is tasked with showing that the software does what it intends. This stage automatically sets up an environment (servers and surrounding infrastructure) that is very similar to the actual production environment, a task that can take up to weeks when done manually [Chen, 2015]. The binaries and other

artifacts created in the commit stage, along with environment configurations stored in the version control system, are installed in this environment and an acceptance test suite is run. The purpose of the acceptance tests is to ensure that the functional and non-functional requirements of the customer and users are met [Humble and Farley, 2010]. If errors arise, the pipeline is stopped and the developer is notified, just like with all other stages of the pipeline.

The manual test stage. Where a continuous deployment pipeline would automatically continue into the release stage, CD pipelines commonly contain a stage where the product increment is manually tested before deployment. Testers can include both developers, dedicated testers and customers or users [Humble and Farley, 2010]. The purpose of manual testing is to catch any bugs that the automatic tests may have missed (e.g. through exploratory testing) and to make sure that the intended value is delivered (e.g. through manual acceptance testing) [Humble and Farley, 2010]. As with the acceptance test stage, effort is reduced by automating the environment setup and notification of instance availability to the testers [Chen, 2015]. If the software meets the criteria of the testers, the increment can be considered a candidate for release [Chen, 2015].

The release stage. The final stage of the CD pipeline consists of a set of actions, usually scripts that package the software appropriately and deploy it into production. This sequence, when ideal, only requires the click of a button when a decision has been made to release [Chen, 2015]. Sometimes, the release stage first deploys the software into a staging environment, an environment identical to the production environment, in order to make sure everything runs smoothly [Humble and Farley, 2010]. A set of last tests, called smoke tests, should be run to check that the application and all the services it depends on are actually up and running as intended [Humble and Farley, 2010].

While the above constitutes a typical CD pipeline, different stages may be added or removed based on need. For example, a performance test stage may be necessary before release in order to give an indication of how the latest change has impacted the performance of the software [Chen, 2015]. Some projects may have separate security test stages [Leppänen et al., 2015]. On the other hand, smaller, less critical applications may not even need a manual test stage before release, but rely on manual testing in production.

2.2.2 Tools

There is no set toolkit for an optimal implementation of CD. What tools and technologies are used depends on factors such as the context of the project,

the existing knowledge and previous experiences of the stakeholders, and the available resources. Some sort of version control system (VCS) however, is mandatory. Typical open source version control systems include Git[1], Subversion (SVN)[2] and CVS[3]. The second aspect of the CD pipeline is the CI software, used for example to fetch code from the VCS, compile the software and run tests. Examples of common CI software are Jenkins[4], Hudson[5] and Go[6]. Setting up environments automatically can be done using custom scripts, but tools like Docker[7] and Vagrant[8] can help with consistency.

2.2.3 Practices and principles

While the technology involved in establishing a pipeline is of utmost importance, it is hard to argue that much can be accomplished if organizations do not adopt the appropriate practices when pursuing CD. What follows is an overview of principles and practices that have been proposed as characterizing successful CD.

Small code changes per release. Discussing CD without the aspect of increased release frequency is pointless. As one of the main targets, the increased release frequency that CD tries to achieve naturally results in reduced size of changes between each release. This is important as it means that less errors can occur in a release, and they will be easier and faster to fix [Fowler, 2013]. In order to increase the release frequency the code needs to be in releasable state more often.

Fast automated tests. Automated tests are a corner stone of any modern software development practice. With CD, fast test suites become of increased importance as they need to be comprehensive enough to guarantee high quality, but simultaneously limit the speed of the release cycle [Leppänen et al., 2015]. Furthermore, long test runs force developers to wait for results, which by default is time wasted. Optimizing and parallelizing tests may be necessary, or developers may even start ignoring the results [Neely and Stolt, 2013]. It has also been suggested to automatically fail test runs that extend past a set threshold, in order to force optimization of tests that take too long[Humble and Farley, 2010].

[1] `https://git-scm.com/` – Open source VCS
[2] `https://subversion.apache.org/` – Open source VCS
[3] `http://savannah.nongnu.org/projects/cvs` – Open source VCS
[4] `https://jenkins-ci.org/` – Open source CI server
[5] `http://hudson-ci.org/` – Open source CI server
[6] `http://www.go.cd/` – Open source CI server
[7] `https://www.docker.com/` – Open source environment tool
[8] `https://www.vagrantup.com/` – Open source environment tool

Deployable software over new features. As with CI, all focus should lie on keeping the build "green". That is, if a code change breaks the build, the responsible developer investigates the issue and fixes it before anyone commits new code to that branch. Not abiding by this practice makes errors harder to find and may lead to developers getting used to "red", or broken, builds [Humble and Farley, 2010]. A red build is not a releasable build and the errors will need to be fixed eventually.

Organizational support and collaboration. The implications of adopting CD do not only concern the development team. Rather, multiple teams and stakeholders need to cooperate in order to make the change successful [Leppänen et al., 2015]. Selling the concept to the involved parties can be hard, but a common ground on which to build the practice can be achieved by convincing functional entities of the benefits that concern them the most [Neely and Stolt, 2013]. At any rate, it is useful to acknowledge the fact that some organizational cultures are less receptive to change than others, which may prove to be an obstacle [Leppänen et al., 2015]. It has been suggested that a DevOps culture is a prerequisite for successful CD, with development and IT operations working closely together [Fowler, 2013].

Build binaries once, deploy the same way. With CD, we want to make sure that all increments that pass through the pipeline will work in production. If binaries are built more than once in the different environments of different stages, we cannot be sure that they are identical and that the ones that make it into production are the ones that were tested and proven to work. Thus, the only binaries that should be released are those that were built in the commit stage. Furthermore, the software should be deployed in the exact same manner regardless of environment. If it is not, there is no way of guaranteeing that the deployment process will work. Configuration files can be used to cover the differences between the environments, but the scripts and the process for all deployments should be the same. [Humble and Farley, 2010]

Deployment by the push of a button. A sign of a sound CD practice is that the current version, including the latest change, of the software can be deployed whenever, by an action as simple as the push of a button [Fowler, 2013]. This requires two things: a green build and automatic deployment scripts. The deployment scripts should be the only way that anyone deploys the service into production, as that makes every deployment auditable and reliable. If some part of the deployment process is manual, the risk of human error is introduced to the release stage. The deployment scripts, just like the rest of the software, need to be maintained, tested and kept in the VCS. [Humble and Farley, 2010]

Information radiators and metrics. One of the main focus points of CD

	[Humble and Farley, 2010]	[Fowler, 2013]	[Neely and Stolt, 2013]	[Chen, 2015]	[Leppänen et al., 2015]
Accelerated value delivery		x		x	x
Reduced risk of release failure	x	x	x	x	
Increased productivity	x			x	
Quicker user feedback	x	x	x	x	x
Improved software quality				x	x
Better visibility of progress		x			x

Table 2.2: A summary of the benefits of CD in existing literature.

is to speed up the feedback on production readiness after a code commit [Fowler, 2013]. In order to achieve fast feedback, not only does the deployment pipeline need to be relatively quick, but the feedback needs to be presented and visible. Information radiators, such as screens on the wall, can be used not only to show the status of a build, but also measurements like cycle time, test coverage and build success percentages [Humble and Farley, 2010]. Feedback from the development process has been perceived not only to support the CD practice, but also to heighten developers' sense of accomplishment and motivation [Leppänen et al., 2015].

2.3 Benefits of continuous delivery

The second research question in this study is about the actual outcomes of the adoption of CD in the case organization. To serve as background for answering that question, this section is a review of the proposed and observed benefits of CD. The benefits in this section are summarized in Table 2.2.

Accelerated value delivery. As the release frequency of the software under development rises, valuable features and fixes can be delivered to the end users much faster [Chen, 2015]. Delivering often can also be seen as a means of producing less waste, as ready features don't have to lie in wait for a planned release, but can be deployed as soon as they're done [Leppänen et al.,

2015]. This is one potential source of an increase in customer satisfaction.

Reduced risk of release failure. With higher release frequency, the changes to the code between releases are reduced. This means that fewer things can go wrong with any one release [Fowler, 2013]. If anything should go wrong, pinpointing the point of failure and fixing it is easier, and the deployment pipeline makes it possible to automatically roll back to a working version [Humble and Farley, 2010]. This benefit has further implications. When releasing is a reliable and practiced activity, the amount of stress amongst developers and other stakeholders is reduced [Chen, 2015; Neely and Stolt, 2013]. Furthermore, the sense of quality and stability can lead to increased trust in the relationship between the developing organization and the customer [Chen, 2015].

Increased productivity. In one case study, the developers and testers spent up to 20% of their time configuring and maintaining the environments used for development before adopting CD [Chen, 2015]. Most of this effort can be avoided with a deployment pipeline that automatically sets up the environments needed for each stage. The initial implementation of a deployment pipeline can indeed require vast amounts of effort, but the ultimate goal is to free up developers' time for actual software development [Humble and Farley, 2010]. Eliminating manual, non-value adding work through automation is no new concept, but is central to CD. In addition to automatic environment configuration, automation of tests can introduce substantial savings to a project [Humble and Farley, 2010].

Quicker user feedback. Several sources have observed the benefit of getting early feedback on the usefulness and value of new features under development Fowler [2013]; Chen [2015]; Leppänen et al. [2015]. Instead of spending large amounts of effort on developing a feature that may or may not be that valuable in real world use, developers choose to abandon its development if users find it useless early on. Thus, the whole undertaking is more likely to result in the "right" product [Fowler, 2013]. Frequent releases also allow for experimentation, as new ideas can be tried out without risking serious losses [Neely and Stolt, 2013]. Moreover, since developers can respond to the user feedback more quickly by releasing bug fixes and new features, the customer satisfaction may be improved [Leppänen et al., 2015].

Improved software quality. Due to the fact that CD relies on a large amount of automatic testing, exhaustive test suites are required. Several projects that have adopted CD report that planned, comprehensive testing, combined with smaller releases results in higher overall software quality [Leppänen et al., 2015]. One organization noted an open bug decrease of over 90% in a project where roughly a third of developers' time was previously spend fixing bugs [Chen, 2015]. Not only is effort reduced, but customers

don't have to wait for a big planned release until the bugs are fixed, as a solution can be deployed as soon as it's done.

Better visibility of progress. Also pertaining to the relationship between parties involved, changes actually being deployed into production makes progress much more trustworthy than just the word of the developers [Fowler, 2013]. One study revealed that the frequent releases made it easier for stakeholder to stay up to date regarding how the project was proceeding [Leppänen et al., 2015].

2.4 Challenges in adopting continuous delivery

As is the case with most development practice, pursuing a state of continuous delivery is not without its challenges. These may not be exclusive to CD, but have all been identified as obstacles in previous case studies on CD adoption.

Complex software. Some software projects consist of many interdependent components or modules, and sometimes they even have interdependencies between other projects. This can cause problems when trying to automate and streamline the deployment pipeline [Leppänen et al., 2015], inherently leading to long integration times often including manual labor. If the components are developed by separate teams, this puts further stress on the transparency, commitment and process awareness of the teams involved [Olsson et al., 2012]. Moreover, the size of the code base may prove to be a challenge. The bigger the size of the code base, the longer every stage of the deployment pipeline takes, which in turn prolongs the potential release and feedback cycles [Leppänen et al., 2015].

Large test suites. For the tests to be able to ensure sufficient quality, they need to be exhaustive. This means that a lot of the developers' time will go into writing tests, which may require new knowledge and attention. A main challenge however, lies with the fact that tests take time to execute [Leppänen et al., 2015]. The challenge of creating fast but effective test suites naturally increases in difficulty as the complexity and size of the project grows.

Legacy code. Projects that start from a clean slate arguably have better chances of successfully adopting CD than those that are already in development. For example, software that has been in development for a long time may not have been designed for automated testing at all [Leppänen et al., 2015]. In this case, moving to CD may be challenging not only from a technical standpoint, but also from the social point of view, as developers have to rethink the way they write new code to be testable [Leppänen et al., 2015].

Environment discrepancies. The environments used in the deployment pipeline should be as similar as possible to the production environment, especially towards the end of the pipeline [Humble and Farley, 2010]. Otherwise, unexpected errors may arise which may be hard to trace and require non-value adding effort. Issues have also been reported to arise when the developers' environments differ from those in the deployment pipeline [Leppänen et al., 2015]. This challenge can be tackled with good configuration management and virtualization, so as to minimize the risk of environment dependent defects [Leppänen et al., 2015].

Customer and domain constraints. One thing to keep in mind when pursuing CD is that not all customers may need or even want shorter release cycles [Leppänen et al., 2015]. This is not a direct obstacle for CD, as not all release candidates have to be released, but may become relevant if the developing organization is striving for continuous deployment. Furthermore, it can dramatically limit the benefits of CD, such as quick user feedback, small changes per release and experimentation. The domain may constitute another challenge. Software that is intended for highly regulated environments (e.g. medical), or contexts where any unscheduled downtime is too expensive (e.g. industrial systems) can make the full adoption of CD almost impossible [Leppänen et al., 2015]. If the deployment environments are very diverse and have differing configurations (e.g. telecom) it can be hard to implement a fully automatic pipeline that covers all the permutations of potential production environments [Leppänen et al., 2015].

Collaboration and transparency. As previously stated, the cooperation of all units within an organization is required to perform CD successfully. This poses a few challenges. First, it may be difficult to effectively provide a view of the project status to all those who need it [Olsson et al., 2012]. Traditionally, CI practices have involved status reports by e-mail or at the very most, on a screen close to the developers. Making development and production status visible to the entire organization is both a technical challenge (how to deliver information) and a data analysis challenge (what information to deliver). Second, it can be difficult to involve and communicate with all the stakeholders. One case reported making the mistake of not involving the marketing and sales people in the adoption of CD, which resulted in those departments having no idea of when certain features would be released [Neely and Stolt, 2013]. Such lack of synchronized ways of working and transparency can disrupt the sales process and be decremental to relationships between departments [Neely and Stolt, 2013].

Change resistance. As with any major changes to established and acquainted behavior, getting an entire organization on board with moving to CD can prove challenging. Several cases have reported that a stiff orga-

nizational culture challenges the adoption of CD [Leppänen et al., 2015]. Furthermore, it has been shown that an organization with a history and tradition of constant improvement and change can quite effectively, although not without other challenges, adopt CD [Neely and Stolt, 2013]. Resistance to change can be an issue both on a personal level, such as a developer being vary of unfamiliar practices that are enforced, and on a decision level, where management may not want to take the risk of losing productivity over new, experimental processes. The roles and responsibilities of development, management, marketing and support personnel change when adopting CD, which can increase the pressure on employees [Claps et al., 2015].

Supplier dependency. Some software projects source parts of the product or service from separate suppliers either within or outside of the organization. These kinds of projects face the challenge of having to coordinate and synchronize the working practices between the units involved [Olsson et al., 2012]. This means that the "weakest link" in the supplier network will set the pace of the entire development effort. Furthermore, slow communication and component integration issues are barriers that organizations may face when adopting CD in a supplier dependent context [Olsson et al., 2012].

2.5 Modeling integration flows

This study uses the extended Ståhl and Bosch notation for modeling software integration flows [Ståhl and Bosch, 2014a]. As the practice of CI started growing wildly in popularity amongst practitioners, the authors noted that the actual practices and the implementations of CI varied largely. Thus, the model was created as a means of describing the software integration flow of a particular case, enabling the direct comparison of different implementations [Ståhl and Bosch, 2014b]. It was later extended and used in an evaluative study by the authors, where the notation was used to successfully describe the software integration flows, or build pipelines, of five out of five cases [Ståhl and Bosch, 2014a]. The Ståhl and Bosch notation was selected for this study on the basis of its proven performance and in order to keep the descriptions of different build pipelines comparable and uniform.

The model uses five notational elements to describe build pipelines, which are summarized in Figure 2.2. *Input nodes* (triangles) are sources which provide data. *Activity nodes* (rectangles) perform one or several actions on data input or other parts of the pipeline. *Trigger nodes* (circles) are used to describe external triggering factors. *Input edges* (dashed arrows) show the flow of data between nodes, while *trigger edges* (solid arrows) describe the conditions for and origins of different stages of the pipeline.

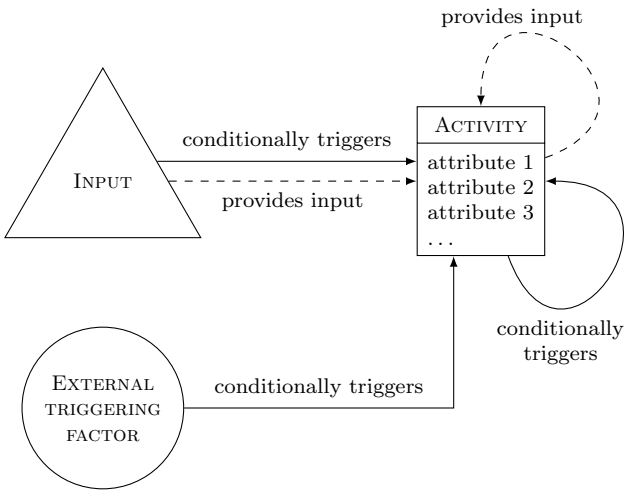


Figure 2.2: The elements and relations of the Ståhl and Bosch 2014a notation

In Figure 2.3, an example of a standard continuous delivery deployment pipeline, similar to the example in section 2.2.1 is presented. Here, a commit to the version control system triggers the commit stage. If all the steps of the stage, defined within the activity as attributes, are successful, the acceptance test stage is triggered. Similarly, the successful execution of the acceptance stage will trigger the manual testing stage. If stakeholders involved in the manual testing sign off the release candidate, the decision of to deploy it will trigger the deployment activity.

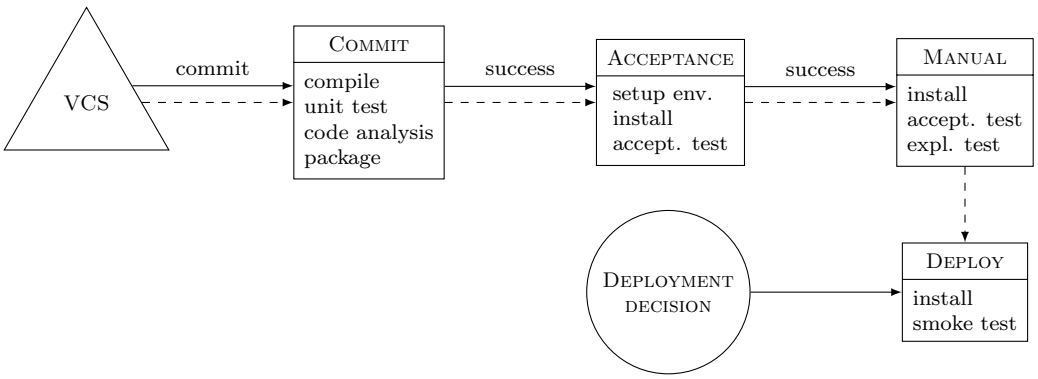


Figure 2.3: Example of a typical CD pipeline using Ståhl and Bosch 2014a notation

Chapter 3

Research Design

3.1 Research motivation

While several studies on the use of continuous delivery have been made, our field is still lacking in empirical, real-world studies of the practicalities CD adoption and its concrete outcomes. Existing research on the adoption of continuity in software development seems to focus either on the technical aspects of CD [Chen, 2015; Bellomo et al., 2014], the theoretical ideal and evolution of CD [Olsson et al., 2012; Fowler, 2013] or the perceived benefits and challenges of CD [Leppänen et al., 2015; Chen, 2015; Neely and Stolt, 2013]. Furthermore, the topic of the context appropriate level of continuity is rarely mentioned.

The developing organization studied in this thesis also has a set of reasons for investigating this matter. As a company striving to improve their value delivering practices, they are interested in knowing whether the effort spent the improvements actually brings any real value. While it may feel like the changes were worth it, it is a difficult task to actually prove it without any data. Thus, we need to know what data to gather and what to measure, but also understand what the data tells us about the state of the practice.

RQ1 How has the case organization been adopting CD?

We are interested in detailing the adoption history and describing what the process of introducing CD practices looks like. The goal is to gain an understanding of the events and actions involved in the adoption, the reasons behind them, challenges and enablers as well as the role of organizational and technical factors.

RQ2 What have the outcomes of the adoption of CD been?

Can the project actually be considered to be in CD mode? Is the adoption visible from the customer's perspective? What benefits have the new practices and tools provided to the developing organization and the customer?

RQ3 How can we measure the success of the adoption of CD?

Are the changes visible in any of the data produced and logged by the tools in use? Does this data support the qualitative results from RQ1 and RQ2?

3.2 Case description

Solita is a Finnish provider of digital services that, in their own words, are "specialized in creating value for their clients by integrating technology, content and business processes". Founded in 1996, they are based in Tampere and have offices in Helsinki and Oulu. At the moment of writing, Solita employs around 450 people across these three offices. Their largest customers include many state-run entities, such as the Ministry of Justice, the National Land Survey, the Finnish Transport Agency and YLE. Large private customers include Sanoma, Finavia, TeliaSonera and Fazer.

The organization being studied develops a suite of applications at Solita for their customer, Tekes. Currently, there are about 13 developers involved in day-to-day operations. While many companies organize their teams around individual products, the developers in this case organize around projects. This is due to the fact that the applications involved are mature and thus highly integrated. Thus, small, 2-4 developer teams are formed for the purpose of implementing a project (a large feature or set of features) that usually demands changes to several of the existing applications. Solita employees have been active in making improvements to the ways these teams work, collaborate, and deliver code over the course of several years, with the purpose of making development increasingly continuous. While this is an ongoing process, this study covers the history and results of the continuous improvement so far.

An theoretical depiction of the organization at the moment of writing can be seen in Figure 3.1. The diagram intends to provide a general understanding of the way stakeholders are organized in the context of the study. Developers organize around projects concerning one or several applications. A single developer may be part of several projects. The applications each have a designated lead user, and are all under the supervision of the IT management of Tekes. Not visible in the diagram is development done outside of

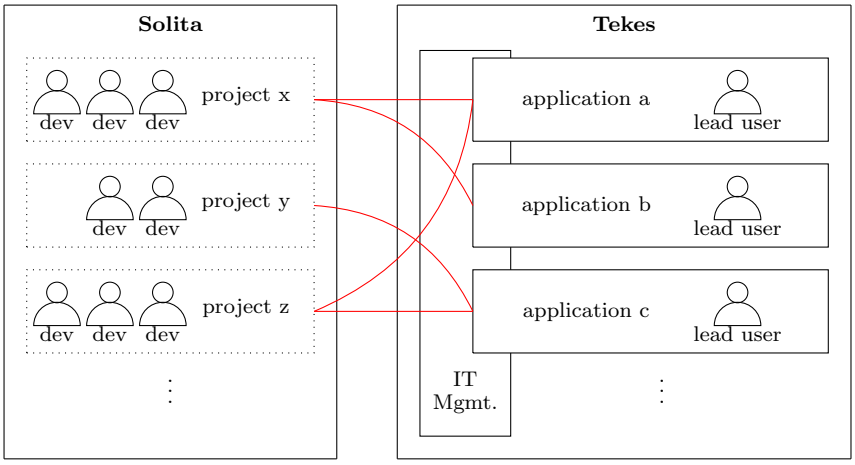


Figure 3.1: A diagram describing the way stakeholders organize around development in the case organization

projects, such as smaller bug fixes.

3.2.1 Application suite

The suite of applications is being developed for Tekes, the Finnish Funding Agency for Innovation. Tekes is in the business of funding research projects, new innovations and their development. As a part of the Finnish Ministry of Employment and the Economy, they employ around 400 people and finance around 1500 business (private) research projects and 500 public research projects each year. The core purpose of Tekes is to fund research and innovation projects that stimulate and improve the Finnish economy, and as such they work as a non-profit organization.

The case organization develops a multitude (16 the time of writing) of applications and systems of varying size that serve the customer in their day-to-day operations. The applications are interconnected and interdependent to some degree, which adds to the complexity of their development. Furthermore, several of the applications have a long history and are weighted down with legacy code and technical debt. All of the applications are developed in the Java language.

The systems under development range from large to small. The most important applications are the custom CRM solution, the ERP system, the registry system, the online errand system and the service bus. The main applications range from around 200k to 400k lines of code in size.

3.2.2 Case background

This subsection is a short summary of the history of the studied case prior to the adoption of CD. It intends to give an understanding of the background of the case and the context wherein the changes have been made. The recent history is detailed in chapter 4.

Prior to 2010, Solita's contract included purely the delivery of one application. Over the years, there had been multiple suppliers and vendors involved in the delivery of software to the customer. Supplier responsibilities were largely restricted. For example, the application vendor was only responsible for providing that particular application, delivering it and building it in an environment where one vendor provided the hardware, another the middleware, a third the networking, and so on. Deployments were done rarely, a few times per year, through what was reportedly a risky, stressful and long process. There were also no rigid service level agreements (SLA's) regarding aspects such as service uptimes that would have encouraged any type of process improvements.

> *"The customer had some internal SLA's for how much the service can be down in a year, and those have gotten tighter. When I started, of course it was sad if production was down but no one was subject to any sanctions or anything like that. Someone just, well not screamed, but was angry."* — Developer 1

The recent history documented in this study does not only describe a push for CD from the developers but also a strive towards increasingly reliable and less fragmented supplier relationships from the customer's side.

3.3 Research method

3.3.1 Data collection

In order to gain a deep understanding of the case and context, interviews were chosen as the main source of research data. Interviews were conducted both with the developing organization (Solita) and their customer (Tekes). The interviews were largely performed according to the standardized open-ended interview format, but with elements of informal conversational interviewing. The format of the interviews was the result of several deciding factors. First, standardized interviews ensure that each interviewee answers the same questions, thus making the interviewees' responses directly comparable [Patton,

2002]. Second, some conversational elements allowed us to explore unexpected but relevant answers and concepts further, by deviating from the scripted questions. Each interview was held with one interviewee and two researchers (me and a colleague) present. The purpose of interviewing one person at a time was to allow the interviewee to speak as freely about the topics as possible. Having two researchers present ensured that all topics were covered, and note-taking could be offloaded on the person not asking questions at the time. In addition to written notes, the session audio was recorded and then transcribed by an external professional party.

In order to get the customer's view of the situation, two employees at Tekes were interviewed. One interview was conducted with an IT Architect, who is heavily involved in the collaboration with Solita, and one interview was held with the lead user of one of the systems, so as to get a more non-technical perspective. The goal of these interviews was to understand how the customer perceives the changes to processes and methods, and to identify how, if at all, the alleged use of CD is actually valuable and useful to their work. The interviews lasted roughly two hours each. The perspective of the developing organization was obtained through two interviews with developers at Solita. These interviews, which lasted about 1,5 hours each, covered the state back when the developer was assigned to the project, the changes that took place since, and the current situation. The developers were also asked to comment on a set of visuals based on metadata from systems and tools used in development. Furthermore, several meetings with a researcher employed at Solita and the lead developer of the project were held, where valuable insight into the case was gained. A summary of the interviews and their themes can be found in Table 3.1.

In addition to the interview data, Solita provided some metadata from several systems and tools used in development for the purpose of quantitative analysis. This data includes the installation logs from the production environment from November 2013 to June 2015 (time, application name, comment and version number), the SVN log for the largest application Eval from November 2006 to June 2015 (time, filenames, author and comments), the release history from Jira for November 2010 to February 2015 (release name, date, description and issue amount) and issue data from Jira from November 2010 to February 2015 (issue types and dates of states, comments and assignments).

Organization	Roles	Themes
Developing org.	Developer 1	General information
	Developer 2	Starting situation
		Changes and improvements
		Current practices
		Selected metrics
Customer	Lead User	General information
	IT Architect	Development process
		Collaboration
		Perceived changes
		Opinions on changes
		Needs and values

Table 3.1: A summary of the roles of interviewed stakeholders and the themes discussed.

3.3.2 Data analysis

Software engineering is largely a social science. Thus, the qualitative information – the knowledge, opinions and memories of interviewees – establish the primary source of findings in this thesis. For the purpose of extracting the important events, reasons, challenges and results, but also any unforeseen information relevant to the case, thematic analysis with an open coding approach was performed on the transcripts of interviews and meetings. Furthermore, the metadata from tools and systems were subject to explorative analysis in order to verify and complement the interview data. This section describes the qualitative approach.

3.3.2.1 Qualitative analysis

Thematic analysis is, as the name might disclose, a way of finding themes, or patterns, within a body of information. Although the method has its roots in psychological research, it is widely applicable to almost any field where text is analyzed, and can be incorporated into many different methodologies and frameworks [Braun and Clarke, 2006]. The purpose of thematic analysis is to reduce a body of text, in this case an interview transcript, into a manageable and rich dataset with a certain level of categorization based on the themes that are discovered. There are two important things to note about this method. First, thematic analysis acknowledges that the researcher plays a central role in the themes that are discovered [Braun and Clarke, 2006]. By not expecting themes to objectively emerge from the data, we accept that

there exists a conscious process, by which the researcher makes decisions on whether or not a section of the data constitutes a theme, that can be audited and reasoned about. Second, there is no specific set of rules that define what is or is not a theme [Braun and Clarke, 2006]. Rather, it is up to the researcher to assess the contextual importance and uniqueness of a piece of information, be it a sentence or half the document.

Thematic analysis builds upon the concept of coding topics in the text and clustering them. Coding is the process of classifying the content of, for example, an interview so as to extract and catalog the relevant topics [Patton, 2002]. In any case, thematic analysis is not bound to any specific coding method. Selecting an appropriate method for the context of the case is up to the researcher. There is, however, a suggested six-step procedure for conducting thematic analysis by Braun and Clarke, that is used in this study. The steps are described in the following list:

1. Familiarize yourself with the transcript(s) and note down ideas.
2. Generate codes for the entire data set in a systematic way.
3. Group the codes into potential themes.
4. Check if the candidate themes match up with the ideas and codes both within the separate, coded extracts and across the entire data set. Revise the themes if necessary.
5. Clearly and consistently name and specify the revised themes.
6. Produce a final report containing extracts that communicate the themes in a compelling way, relating to the existing literature and research questions.

Open coding is a method that lends itself well to thematic analysis. It too is an iterative process where each step takes the researcher closer to the final understanding and interpretation of the data. Open coding is an analytical process, by which the relevant concepts and their characteristics are discovered in the data [Strauss and Corbin, 1998]. When performing coding, we look for concepts within the text, and compare these to each other in order to understand relations between phenomena and groupings of concepts. In a way, one could argue that the methodology in this study is semi-open, as a set of categories of special interest already existed; we wanted to look at what happened (e.g. events, changes, incentives) and what the results were (e.g. benefits, challenges). However, the coding was performed in an open manner, i.e. unexpected phenomena and concepts were

not discarded. There several ways to perform open coding on varying levels of detail: line-by-line, by sentence or paragraph and by document [Strauss and Corbin, 1998]. In this study, the transcripts were coded by sentence or paragraph, as the interview format only allowed for a few concepts to be presented in each answer.

All in all 9 themes and over 120 codes were identified. The selected themes were, in no particular order: methodology/process, collaboration, testing, deployment pipeline, monitoring, productivity, software quality, work satisfaction and agnosticism.

3.3.2.2 Quantitative analysis

All the metadata provided by the case organization (Jira, SVN and installation logs) was subjected to explorative analysis in order to evaluate the qualitative results. The analysis is derived from the identified characteristics and benefits of CD. Some data turned out not to provide any added value, and was discarded. The SVN log could not be reliably used to analyze commit sizes, as lines of code (LOC) sizes were not available. Furthermore, the SVN log was only limited to the repository of one of the applications. The installation logs turned out to contain many redundant entries and other unreliable information. As every installation for each application was logged separately and some installations appeared to have been rerun within seconds and multiple times, a more reliable source of release and deployment data is the Jira project. This being the case, only metadata from Jira is included in the results of this study. There is certainly room for additional analysis of richer datasets in this field.

The data from Jira was largely left unmodified, although some parts of it needed to be filtered. The release issues contained application version entries that were released and deployed to production along with version entries that never actually ended up in production. For example, when the developers worked according to a Scrum-like model, each sprint resulted in a "sprint release" in Jira. However, no actual deployments were made until three sprints were done, at which a "backlog release" was created. This constituted the actual release.

Furthermore, the practice used over the entire investigated period has been to split larger issues into smaller issues of roughly the same size. Thus, so called "parent issues" have been ignored in the analysis in order not to distort the results. Also, only issues that have actually been decided to be implemented are included.

The final data sets were analyzed using the R programming language. Performance indicators that should reveal whether or not the adoption of CD

had provided any significant benefits were selected and then calculated based on the quantitative data. This was largely an iterative process performed together with the case organization. Selecting such time series that would be appropriate for the level of precision in the data was an essential part of the analysis. Too short intervals would capture the inconsistencies in tool usage rather than give insight into the actual development practices.

Chapter 4

Adopting continuous delivery

In this chapter, the adoption process is detailed as a series of challenges and activities that have resulted in the situation at the time of writing. The purpose is to describe how the organizations involved have pursued CD, what challenges they faced and by which measures these challenges were confronted. Direct quotes from stakeholders are used when applicable to give the reader richer insight into the case. Evaluation of the success of these actions is left to chapter 5. This chapter begins with the initial situation recognized by interviewed stakeholders, after which adoption activities and results are presented in sections according to the themes identified during thematic analysis. For a chronological overview of the events described in this chapter, see Table 4.1.

4.1 Initial situation and challenge

Although the supplier-customer relationship has its roots in the year 2004, Solita won the contract for development of the core applications at Tekes in the beginning of the year 2010. Thus, it seems only fitting to start investigating the adoption of CD at that point in time. Back then, about 4 developers were involved in the account. The development mode largely adhered to an undefined iterative waterfall model. Heavy specifications were written for each new feature by the customer, delivered to the developers, developed in relative isolation and finally tested. This is where the first challenge identified in this study arose: a customer seldom knows what they need before they get it. As will become obvious later in this chapter, especially developers, noticed that even if the end result corresponded with specifications, the implementation would still require changes. Developing against rigid specifi-

2010	Solita wins contract for development of all of Tekes' core applications
2011	CI-server (Hudson) is taken into use
2012	Nightly dumps of production databases for development use
2013	Build tool switch (Ant replaced with Maven)
2013	Database migrations are automated
September 2013	Application server migration initiated (WebLogic replaced with JBoss)
December 2013	The first deployment scripts are taken into use
January 2014	Separated environment configurations enable environment independent builds
February 2014	All applications have deployment scripts
November 2014	Server configuration is automated (Ansible & Vagrant)
December 2014	Data center is moved
2015	Successful commits trigger deployment to customer acceptance test server (trial)
2015	Customer environments are monitored and smoke-tested (Dataloop.io & Smokemonster)

TABLE 4.1 Timeline of notable events during the CD transformation.

cations that developers have not been involved with before they are delivered can have several negative effects, some of which have been identified in this case. Primarily, it lead to considerable amounts of wasted work, as ready features would have to be revised after it was discovered that they did not fit the needs of the customer.

> *"The Word documents where it was specified what we were going to do never correlated with reality [...] so the one who really understood the case realized maybe two months later that this shouldn't be like this at all. Either the specs were wrong or the implementation of the specs was wrong."* — Developer 1

Second, this method did sometimes lead to unnecessarily expensive solutions to the customer's requirements. Developers could have provided valuable insight into what the most affordable and effortless way to solve a problem was, based on their knowledge of the software architecture and other technical aspects.

> *"When you have an old, big system it's quite difficult, a simple looking change can be quite immense. [...] It's better to capture what goal we are trying to achieve and then discuss how we can achieve it."* — Developer 2

This challenge was tackled with several changes to the development process. First, and perhaps most important, by establishing a more continuous dialogue and collaboration between the customer and the vendor. This is a central enabler of CD that has been documented in several earlier studies [e.g. Leppänen et al., 2015; Neely and Stolt, 2013]. In the following sections, this initial challenge of developing against specification plays a central role. Several further challenges and corresponding mitigating actions that were implemented are detailed. Section by section, the transformation is described according to themes of improvement that arose from the interviews. We begin with the collaborative improvements, continuing with methodological changes, the implementation of a deployment pipeline and environment monitoring and ending with testing.

4.2 Collaboration

The increased and improved collaboration took several forms, and was the result of multiple factors. An important change was that the developers started communicating with the customer immediately after a need was identified,

and worked together with their stakeholders (users and operations) to find the best solution to a problem. Moving towards this goal is still an ongoing process which can not be considered done, but the practices now look considerably different from those five years ago. By doing so, the rigid specifications could be almost entirely abandoned, as development and specification can be parallelized and requirements extracted through discussion. Now, in theory, development could start almost immediately after the basics of a change were understood. However, transitioning towards such a mode is not without challenges of its own. For example, change resistance and incompatible customer processes can retard the adoption of an approach that seems more uncertain.

> *"With a sped up process, waking up the customer is one [challenge]. [...] Management, too, has to understand that they can't order change entities according to the waterfall model like before, but that being more agile means an actual uncertainty in delivery dates and content."* — Developer 2

Over the years, accommodating development based on continuous collaboration rather than specification documents required active work and dialogue on behalf of the developers, but also considerable effort from customer stakeholders. One important enabler of this change was the fact that people existed within the customer organization that could champion and promote the idea of continuous collaboration and agility both towards users and management. Without this kind of engagement, many of the changes documented in this study would likely have been much harder to implement.

Another meaningful change was the introduction of regular co-located development and collaboration days. Once a week, developers gather in the customer's shared workspace where developers and customer stakeholders can plan changes face to face and arrange workshops on more difficult topics. As Solita employs developers in both Tampere and Helsinki, this is an opportunity not just to close the gap towards the customer, but also between sites. The lack of co-location has also been alleviated by supplying developers with keycards to the customer's facilities and establishing an instant video link between Solita's offices.

When discussing the emphasis on communication, it is imperative to mention the toolset that allows collaboration to be performed efficiently. Here, several tools and improvements to their usability stand out. Issues (tasks) are usually tracked with some type of ticketing software, where issues can be reported and the status of their development updated and monitored. In this case, the two organizations remained slightly siloed by the fact that they used different systems for ticket management. The customer maintained

their own, company-wide ITIL[1] based tool, where incidents were reported and escalated into change requests (CR's). A CR then required an effort estimate by the developers after which it could be pulled into Solita's internal Jira[2] project. The barrier between these two tools essentially caused a lack of transparency between the two companies' processes. As a remedy, they decided to move to only one, shared Jira project to track and plan development.

Tools play an important role when it comes to instant messaging as well, especially in distributed development. This was a central topic brought forth by all interviewees. In the spring of 2014, both parties switched from using a federated Lync[3] solution to Flowdock[4]. This was largely due to the fact that federated Lync with suppliers was no longer contractually allowed with a new infrastructure provider, but as it turns out Flowdock provides opportunities for richer communication and functionality that is more relevant to software development, which we will return to in chapter 5.

4.3 Methodology

As mentioned in the introduction to this chapter, the customer was accustomed to their applications being developed largely in a waterfallish manner when Solita took over the bulk of development. Several stakeholders saw the issues with this outdated way of developing software efficiently, and the decision was made to try out Scrum. However, the findings indicate two challenges. First, there is little evidence that the Scrum methodology was implemented any further than scheduling development according to defined sprints and arranging the required Scrum ceremonies. Secondly, the organizations quickly realized that even with this new approach, using three month-long development sprints followed by a stabilization sprint and deployment, delivery of new features and fixes still was not fast enough.

> *"This was essentially a working practice, but once every three months into production, that's too slow."* — Tekes Architect

[1] `https://www.axelos.com/best-practice-solutions/itil` – Framework for IT Service Management (ITSM)

[2] `https://www.atlassian.com/software/jira/` – Software for issue and project tracking

[3] `https://products.office.com/en-us/skype-for-business/online-meetings` – Instant messaging software for the Windows operating system. Currently known as Skype for Business.

[4] `https://www.flowdock.com/` – Multiplatform software for team collaboration

The realization that some smaller features and bug fixes should be released more urgently led to the next evolution of methodology. The teams introduced Kanban in parallel with their Scrum practices, in order to accommodate faster releases of smaller features and necessary non-functional development. Shortly after this, Scrum was abandoned entirely, leaving the developers to work purely with Kanban driven development. At this point, the basis for the current methodology was born. The amount of developers involved with the account had been growing, so there was a certain need to specialize and focus their efforts. Furthermore, it was noted that many of the smaller features and improvements lost the game of prioritization to large new development initiatives.

> *"The small development tasks were left behind as they were small and less significant, so they were left hanging in favor of the larger projects. They were not getting done, they were just pushed forward. Then we started to realize and wonder about the fact that these same tasks and some new things have been here for a year or two. [...] [The users started complaining] that why can't you fix such a small thing?"* — Developer 2

Thus, the developers started to organize themselves around project lanes, handling larger feature entities, while allocating a few resources to continuously developing smaller development and bug fixes. Note that the high coupling of applications in the suite does not allow simple organization around individual applications, as new changes in general require the modification of multiple applications. A representation of this methodology is depicted in Figure 4.1, and a closer look at project composition can be seen in Figure 4.2. A project typically results in a release and deployment, and can be seen as a collection of changes relating to some themed goal. A project has a certain length (polygon length) and size (polygon height). Note that, for the sake of illustration and simplicity, the example states used in Figure 4.2 are not the only states a change or issue will pass through. In reality, a single change will be internally reviewed, and sometimes deployed as a prototype and revised many times before it is finally considered done.

As evident in Figure 4.1, projects are not developed in parallel. There have been, and still are, challenges preventing the concurrent development of projects. First, this has previously been a limitation of the deployment pipeline, which did not support parallel CI environments for each branch. More on this in section 4.4. Second, as each project is developed in its own SVN branch, integration issues are prone to arise due to parallel development in long running branches. The current approach on mitigating integration

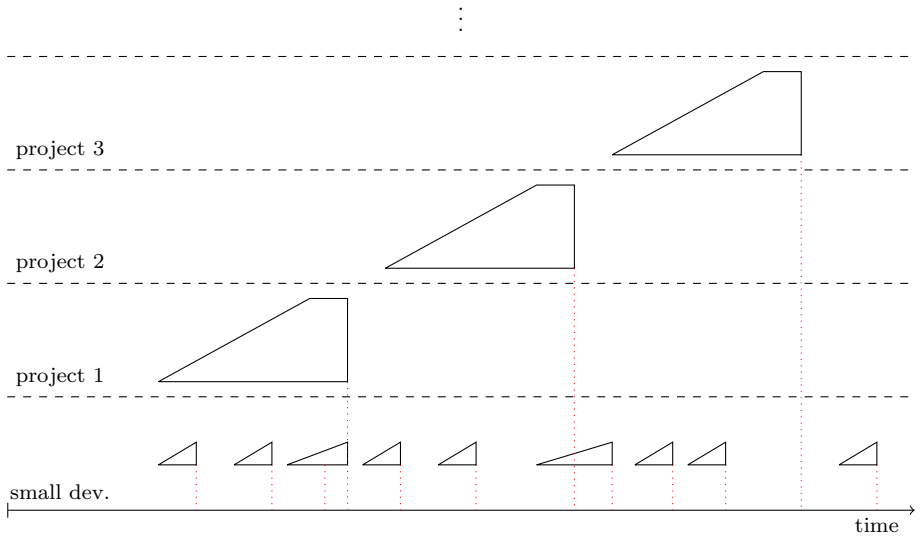


Figure 4.1: A theoretical representation of the Kanban practices used in development. Polygons indicate change entities and vertical lines mark releases. See Figure 4.2 for a visualization of change entity structure.

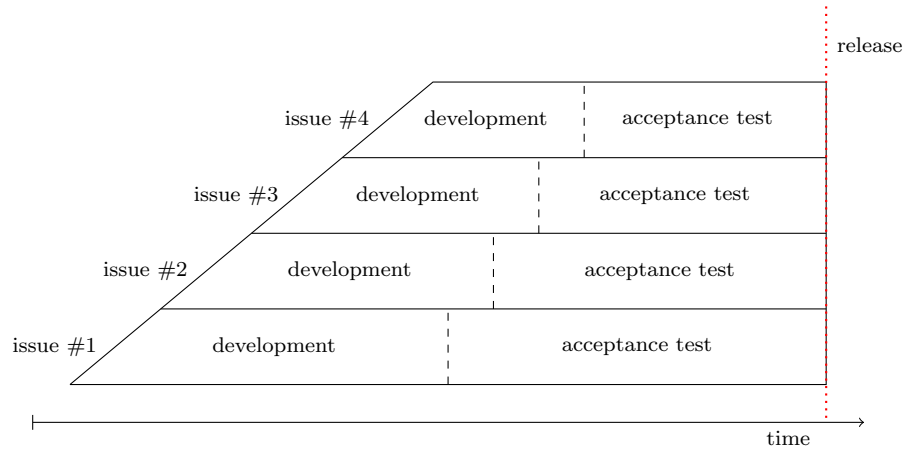


Figure 4.2: The composition of a change entity, or project, as portrayed in Figure 4.1.

issues is to attempt to minimize the project size, by developing new features with a minimum viable product (MVP) mindset. Here too, the legacy code base and high coupling continues to be a major hurdle. As a third way of pursuing parallel development, components that do not impact the same parts of the suite as other planned or ongoing projects are developed beforehand. These entities are then deployed as "preparatory" releases. In the long term, the ideal is to perform all development in the trunk of the VCS while hiding incomplete features behind feature toggles.

> *"It requires a lot of work to be able to do it in small pieces that can be deployed into production. [...] Of course we try to divide it into as small entities as possible, but it's kind of really hard work to get it small enough."* — Developer 1

> *"The kind of development mode, towards which we should strive, but we're not there yet, is to only develop in the trunk and put things behind a switch, so that we can set the toggles on for the environments where a project is developed and set them off in production."* — Tekes Architect

Another essential change to the development methodology was to introduce prototyping of new functionality. The ability to prototype was largely enabled by the decreased deployment threshold, resulting from improvements to the deployment pipeline and automation tools, but also by the now considerably closer and more continuous collaboration between the two companies. This was another attempt at mitigating the risk of developing the wrong solution and wasting effort due to the lack of insight into actual customer needs. The decision to collaborate on requirements served the purpose of bringing the developers closer to the context of use, while prototyping intends to bring the users closer to the actual development. Now, prototypes, and later test releases, of projects under development are deployed to separate customer environments every so often. The lead user of one of the core applications estimated that she receives a new prototype about once every two weeks.

4.4 Deployment pipeline & monitoring

While it may be viewed as a purely technical tool for delivering software to the users, the deployment pipeline is also an important part of both information flow and everyday working practices. In 2010, Solita worked on the account with hardly any automation, neither testing, integration or deployment. The deployment practices were particularly interesting. For each

release, environment dependent artifacts would be built and then transported to the customer on a flash drive, where the applications would be deployed on location. This, of course, is quite a large obstacle when striving to decrease delivery time. Partly the arrangement was a product of the fact that Solita and the suppliers before them were purely commissioned as application providers, and carried no responsibility from the middleware down. This changed in 2010 when Solita's role grew to service provider.

> *"The role of the application provider was to transfer a .war or .ear file to the server, put it in the deployment folder, and then automatic deployment happened at night when the application server service rebooted during the maintenance window."* — Tekes Architect

Another contributing factor was the high security needed in a public sector company handling sensitive data.

> *"The data security limitations of Tekes was a big obstacle. They had to be opened up by Tekes, and of course we always wish that everything would be easy, that no jump servers or VPN:s or anything that blocks access."* — Developer 1

The shift towards an increasingly agile process started with the implementation of a CI pipeline for the purpose of automating integration in the development stage. Adoption took place in 2011, and Hudson was selected as the tool for the job. This first iteration of the pipeline did not yet support either parallel CI nor automation of deployments. Rather, the practice largely adhered to the typical CI presented in section 2.1.

The next steps were to start the homogenization of environments and development of deployment scripts, central aspects of a characteristic CD pipeline. As mentioned in the background to this study, ideal CD requires very production-like environments available as parts of the pipeline, at least as we get closer to the end of the pipeline. Furthermore, we want to be able to reliably and effortlessly deploy any version of an application to the customer's test and production environments. In 2012, Solita introduced scheduled sanitized nightly dumps of the production databases, and made scrambled versions of these available to the developers. This meant that developers could now run their changes against real world data without compromising any sensitive data. Most integrations with other systems are also available to the developers, but some internal services and third party integrations had to be replaced with mockups, as they are only available inside Tekes.

> *"The development environment on our development machines is basically a fully fledged environment, but not quite, as some components are missing. Some are such that are only available within Tekes. There is no single sign-on framework, no document management system and so on."* — Developer 2

The year after, in 2013, they made the switch from using Apache Ant[5] as a build tool to the more modern Apache Maven[6], which in some senses is better suited as not only a build tool but also a build manager. The adoption of Maven was a step towards being able to consistently manage the different versions and releases of an application.

Deployment scripts were the focus along with a large platform transition in the rollover from 2013 into 2014. In early 2014, when all applications had deployment scripts, the threshold to deploy was already significantly lower than three years earlier. The CI server handled unit and integration testing automatically after each commit. When a feature was ready and inspected by another developer, it could be almost instantly deployed to one of the customer's test environments for acceptance testing through a simple series of commands. Deployment to production was not much more difficult from the developers perspective, but required organizational and administrative preparations.

During about the same time, effort was put into making builds environmentally independent. Previously, the applications would have to be built separately for each environment, a time consuming and risky process. This required separating environment configurations from the applications, keeping only environment independent code as part of the build job.

> *"We didn't want to spend time building the right packages daily and then find out in deployment if it works. [...] At some point we got to the point where the same binaries go to all environments and we don't have to do environment specific builds."* — Developer 2

A natural evolution of environmentally indifferent builds was automating the environment configurations. By the end of 2014, a solution was in place where a developer would simply edit a single text file to define which version of the application they wanted deployed on which environment. An internal Jenkins server at the customer would poll this configuration file and the

[5] `http://ant.apache.org/` – An open source build tool

[6] `http://maven.apache.org/` – An open source build manager for Java projects

necessary deployments would be made. Additionally, Ansible[7] and Vagrant[8] were employed for the purposes of automatic server configuration and server virtualization respectively. These two tools came to be used in the entire pipeline, in order to manage configurations consistently throughout the path from development to production.

At the time of writing, trials of full continuous deployment to the test servers are being made with one application. This means that each commit that makes it all the way through the pipeline ends up in the acceptance test environment at the customer without any manual interference. Making this a common practice is not without its own challenges though.

> *"This is a bit like a prototype. It's the direction where we want to go, but like I said earlier the old systems [have some complicating aspects] so they cannot be fully automatically deployed."* — Developer 2

The current deployment pipeline is presented in Figure 4.3. Two things should be noted about the level of automation in this setup: the success triggers marked with stars. First, integration test success rarely triggers the UI test stage, since this would make the pipeline too slow. Instead, the UI test stage is scheduled to run nightly. More on this in section 4.5. Second, automatic deployment to test environments is only implemented for one application, as mentioned previously. For the bulk of the applications, this stage is triggered by a deployment decision, just like deployment to production environments. Production deployment decisions are made together with the customer.

> *"When [the new version] has been accepted and reviewed, we start to look for a suitable evening for a deployment window. [...] Our SLA states that the [core applications] are usable between 7:00 and 17:00 so outside of that we can have short service breaks and deploy to production with prior notice."* — Tekes Architect

How the results of developer activities along the deployment pipeline are monitored is important. Without any feedback on the success or failure of an action, the deployment pipeline is simply not useful. Solita currently

[7] `http://www.ansible.com/` – Open source solution for deployment, config. mgmt. and orchestration.

[8] `https://www.vagrantup.com/` – Open source tool for building virtual environments

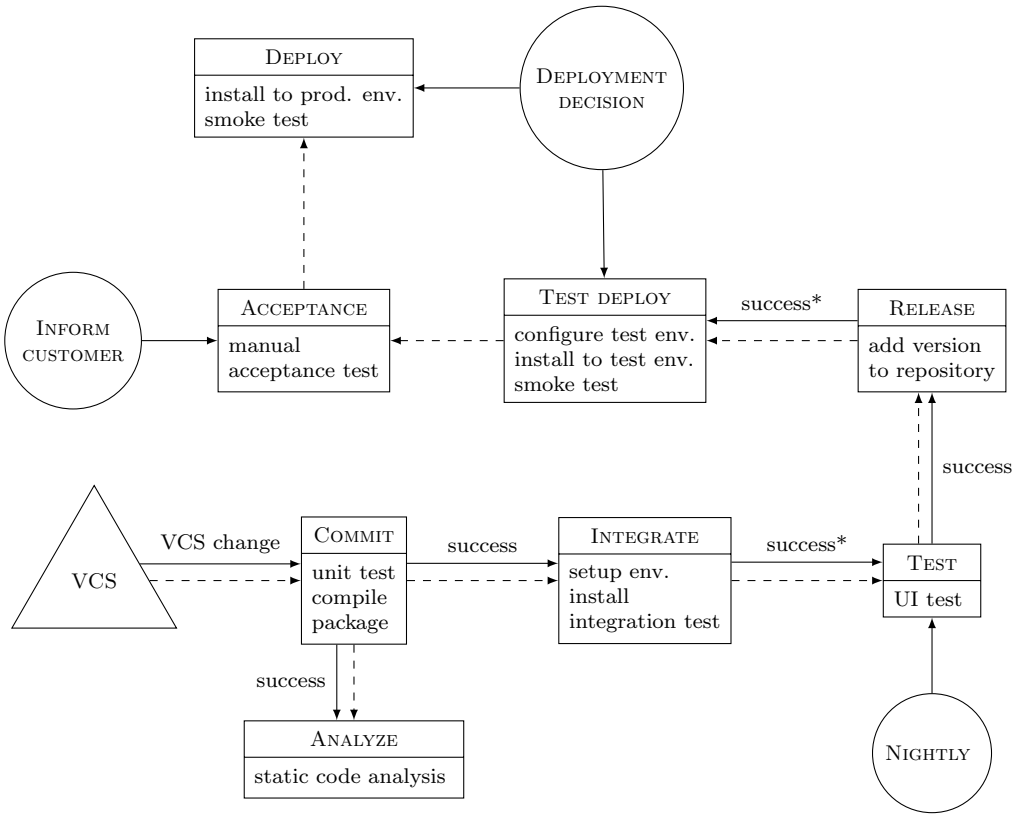


Figure 4.3: The current deployment pipelined as depicted using Ståhl & Bosch (2014) notation.

monitors feedback from most systems in the pipeline. First and foremost, they have put up displays in the team rooms that keep the developers up to date with feedback results from the CI pipeline. These displays report which builds passed and which failed the integration and testing performed automatically after each commit. Instant messages are also sent to the developers on build fail. Furthermore, adjacent displays and SMS messages report on the status of the customer's environments.

> *"For data security reasons we only receive zero/one data [on the status of the environments], Or, well, we do get a text message as well. [...] Each of our core applications are polled with a certain interval [...] and the previously mentioned radiator view turns red if something is broken."* — Developer 1

4.5 Testing

Trust in a deployment pipeline requires testing at a level where we can confidently deploy software after the defined test stages have been passed successfully. As this study concerns continuous delivery, we are interested in proper automated test coverage as well as the effectiveness of manual testing. Back in 2010, there was no clear division of responsibilities regarding testing. Neither did either party really have any knowledge of what the other was testing or plan any of the testing together. Taking testing to a level suitable for CD can be a challenge for a new supplier.

> *"In that mode of operation, acceptance testing [was an obvious challenge]. The quality of testing in general. We didn't have any insight into what Tekes were doing in their acceptance testing. In the beginning we did regression testing like with a new system, that we didn't really know how it worked and in a domain that we didn't understand too well. [...] This manifested itself e.g. by having to do a lot of fix releases after deployment because we hadn't really figured out if it works or not in the testing phase."* — Developer 2

The problems with blind acceptance testing and test coordination was solved by specifying which tests were performed by the developers and sharing this information in the common Confluence[9] Wiki page. Thus, the issues

[9] `https://www.atlassian.com/software/confluence/` – Team collaboration and organization tool

were largely alleviated by the adoption of a more collaborative way of working. By knowing what the developers are testing, the customer can focus on edge cases and avoid redundant testing.

Furthermore, while some unit and integration tests existed, the test coverage of these was not of a satisfactory degree. To this date, no major effort has been made to increase coverage of the code base that was adopted in 2010, it has simply been concluded as a waste of effort considering the limited potential value from such a major undertaking. Instead, the strategy has been to write sufficient tests for all new applications and features. The aim has been to keep the test coverage trend positive, while trusting that the underlying older code still works as intended. Also, user interface (UI) tests have been employed in the deployment pipeline for the applications that have a front end. The UI tests are conducted with Selenium. Static code analysis has been recently introduced as part of the pipeline, but has been identified as a problematic practice when developing against a legacy code base. The static code analysis tool, Sonar, would report errors or warnings caused by the fact that different applications, or parts of them, were developed according to different patterns and sometimes make use of deprecated but working code. The policy has been to fix critical issues reported by Sonar and keep the code quality trend rising.

The way testing is performed by the pipeline can be seen in Figure 4.3. Developers normally perform unit tests and integration tests locally first, before committing, along with some manual exploratory testing. Then, unit tests, integration tests and static code analysis are run with each commit to the VCS by the CI server. UI tests, however, are considered to be too heavy and long running to be performed on each commit. As they can block the pipeline for other commits, and the developer would have to wait a long time for the feedback, the UI tests are run nightly by the CI server.

At the customer's side, after deployment to a test environment, the first thing that is performed is smoke testing. This is another recent development, carried out by an internally developed tool called Smoke Monster. Currently, the tool only performs shallow smoke testing, making sure that the applications are responding and reporting their status through the Dataloop.IO[10] service. When the lead user (similar to the role of a product owner) has been informed of the new version, they perform manual acceptance testing of the changes. The manual testing can take from a few hours up to several days depending on the size of the change and the schedule of the stakeholders. To make this process faster, Solita have developed another tool specifically for making acceptance testing easier. Testiapina (which translates to test

[10] `https://www.dataloop.io/` – Subscription based metrics and monitoring tool

monkey), allows the user to quickly pre-fill fields in long forms based on the output of another application or defined test cases.

The next planned evolution to the testing practices is to employ security and performance testing. Both the developers and the customer have expressed a clear need for automated testing of security so as to increase trust in the deployment pipeline. Furthermore, there have been issues with performance over the years that have been difficult to analyze due to the multi-vendor environment. When performance is suddenly degraded, the investigation process has to take into account all potential causing factors, such as the underlying architecture, infrastructure and networking. Automated performance testing has been expressed as necessary to catch issues caused explicitly by changes to the applications, as soon as they arise.

There is a clear strive to make testing fully automated at some point in the future. This would allow fully continuous deployment without manual testing by either party. However, the consensus is that this cannot be achieved as of now. The technically minded stakeholders state the limitations imposed by the legacy code: the effort to automate testing of the current applications would simply be to large and would block all feature development. The user perspective is that someone with domain knowledge has to go through the changes to make sure that they are up to par with both requirements and legislation.

4.6 Summary and current situation

The most obvious way by which the new working practices can be recognized is how communication takes place within the project. The new process (Kanban, small development lane and platform improvements) is enabled by instant discussion, access to information on customer environment status and potentially instant acceptance testing after development. Co-located days have grown relationships and thus lowered the threshold to open discussion. Organization around development tasks is implicitly decided and done according to need. According to interviewees, continuous improvement of working practices has grown to be a part of everything that is done. On the technical side of things, new tools are used to improve testing practices, alleviate integration issues and automate deployment and environment configuration. In the next chapter, we will take a look at the benefits that stakeholders perceived to result from this transformation.

Chapter 5

Benefits of continuous delivery

The supplier has an implicit impression that the changes and improvements detailed in chapter 4 have resulted in benefits for both the developers and the customer. In order to validate this belief, we interviewed stakeholders from both sides to find out what the perceived benefits of the changes to processes and practices actually were. This chapter is divided into two parts accordingly. First, the developers' point of view is presented, after which the benefits identified by customer stakeholders are described. This division intends to achieve a separation of concerns; some benefits identified by one party may benefit the other party but can only be attributed to the opinion of the source. A summary of the identified benefits can be seen in Table 5.1.

5.1 Developer benefits

The opinions and observations leading to the results presented in this section have been presented by three developers, of whom two have been officially

	Recognized by	
	Developers	Customer
Increased productivity	x	x
Improved collaboration	x	x
Increased quality		x
Reduced risk of release failure	x	
Organizational agnosticism	x	
Improved developer morale	x	
Infrastructural agnosticism	x	

Table 5.1: Overview of preceived benefits and their sources.

interviewed. One of them has only been working on the project for three years at the time of writing, while the other two have been part of the team at least since 2010.

5.1.1 Increased productivity

Increased productivity is one of the main documented benefits of CD and it was discovered to be present in this case as well. As the level of automation increased over the years, developers could focus more on the actual development of the applications and less on menial, repetitive tasks related to operations. However, it has to be noted that much effort has gone into establishing the new tools and practices. Developers agreed that some changes have required more time and effort than initially expected, mostly due to the fact that many of the tools and techniques were unfamiliar to them when adoption started. When asked if the changes were worth it in the end, the unanimous response was positive.

Not only has the automation of tasks decreased the amount of non value creating work, such as manual deployments and configuration, but the time spent waiting for deployment decisions, system feedback and customer communication has been significantly lowered by the new collaboration practices and tools.

> *"Before, it used to be that there was no way of knowing if it would take a week to get it deployed. Now the norm is that when the last commit is done and it's gone through CI, you update the version number in one file and it's done. The default is that it's deployed and you only have to wait for the SMS that it's been installed and then you tell the customer in the chat that they can test it."* — Developer 1

Another major factor contributing to increased productivity is the revised, more agile way of specifying changes. Effort that was previously wasted developing against badly defined or misinterpreted specs is now largely avoided through continuous discussion and prototyping. These practices have also lead to the further benefit of having to rush out less fixes and corrections immediately after deployment of a new version.

> *"[When a user actually tries it out] they say that this isn't how it should be even though they themselves wrote that this is how it should be two months ago. It's sometimes hard to get there but the quick prototype that took two days can save us three months of work or something."* — Developer 1

5.1.2 Improved collaboration

While the increases in speed and amount of communication proved to lower redundant work, it was also regarded as a benefit in and of itself. For example, the continuous discussion on development decisions and specifications is perceived to have improved transparency, as developers started providing continuous input on the viability of different solutions.

> *"The customer was a bit annoyed with the previous supplier as they didn't really have insight into what they were doing. We have tried to improve that so that it's easier for the customer to know what we are doing. For example, why does it cost a hundred thousand to add a button there, when it only costs a thousand to add the button on a different page?"* — Developer 2

Collaborative improvements arose as benefits when discussing almost any theme during the interviews. One such theme was the feedback on deployed changes, which now arrives notably faster than five years ago. Developers also have a much lower threshold for contacting the customer regarding smaller issues or questions during development. Another example is the communication around coordination of testing responsibilities. It has become much more straightforward for developers to decide what tests to write and what to manually check after making changes.

5.1.3 Reduced risk of release failure

Manual steps in any part of the development process introduce risk of failure due to human error. Four factors were identified as having decreased release and deployment risk significantly. First, the environment independent builds have mitigated both the risk of faulty builds due to erroneous manual configuration, and the impact of environment changes on existing releases. Second, the automatic testing of all changes has been proven to significantly reduce the risk of uncovering bugs in the applications after production deployments. Third, the fact that the amount of steps needed to access the customer's environments has been reduced means that there are fewer things that can go wrong in any procedure requiring such access, for example deploying to production or making changes to the environments. Last, developers now have access to environments quite similar to those in production, which has been argued to reduce the risk of errors that can be discovered only once the software is running in the production environment.

5.1.4 Organizational agnosticism

The automation and deployment pipeline has reached a stage where any developer can deploy an application to the test or production environments. Thus, no longer is a vast body of contextual knowledge needed in order to build and deploy an application. The level of mission critical tacit knowledge in the developing organization has decreased. In essence, the impact of the potential loss of a developer to e.g. another project, while still negative, is not as great as before the current tools and processes were put in place.

There is also less need for specialized roles amongst developers. Interviewees admitted a lack of defined areas of responsibility when asked about their and other's roles. No need for explicit roles has been expressed either. Responsibilities are largely temporary and are administered according to the current interest of the developers. This mentality goes along well with the principles of a DevOps organization, which have been seen as a benefit when pursuing CD.

> *"We have tried to break the silos and make it so that everyone gets to do a bit of everything, and divided it more according to ways of working. [...] We have strived for that if someone has knowledge and interest then we let them do that kind of work."*
> — Developer 2

Furthermore, the increase in automation, the new communication practices and many other smaller improvements has allowed the developing organization to grow without larger organizational obstacles. Making constant improvements to the ways of working has proven to enable a scalable team in this case.

> *"Yes, [the effort has been worth it]. Not even a doubt about it. With the old ways of working we wouldn't have been able to grow the team in the way we have. We started off with just a few dudes. Now that we are like twelve it would be total chaos with the old model."* — Developer 2

5.1.5 Improved developer morale

The feedback from developers largely related to a better, less stressed out working environment. Much of the reduction in stress levels results from the reduced risk of failing deployments and the additional work that those require.

> *"It used to be that when something went wrong with a deployment we started fixing it or spent the next day fixing it or something. From our perspective, it sure is terrible if we can't confirm before that, that what we are deploying is likely to work in a way that not everything grinds to a halt over there. A kind of increased peace of mind is definitely an important [reason for the changes]."* — Developer 1

Another benefit in this category is to not have to deal with repetitive, menial tasks. Interviewees reported that many of the ideas that led to changes simply stemmed from individual developers being annoyed with having to perform some task manually, and the improvements were often decided upon collectively after such an issue was brought to light. These kinds of bottom-up changes have resulted in increased developer satisfaction.

There are probably few things more frustrating to a developer than developing something entirely useless. Two important changes towards CD have impacted morale positively in this regard. First, being part of the specification process and being able to provide input on design decisions through continuous collaboration with the customer is unanimously regarded as a positive development by the developers. This allows them to feel that they have a say in defining their own work, and also helps avoid the disappointment of developing something that ultimately does not correlate with the needs of the users. Second, the continuous prototyping, enabled by the deployment pipeline and related automation, allows developers to get faster validation of their interpretation of the requirements.

5.1.6 Infrastructural agnosticism

The separation of environment configurations from the application source, along with virtualization and automation of environment configuration and deployment was reported to have a positive impact on the readiness for infrastructural changes. In fact, large parts of the automation that exists today was developed in preparation for the first data center and infrastructure provider migration in 2014. These improvements mean that the entire project is less dependent on infrastructure providers, which allows the customer to more freely select a more competitive or suitable provider if needed. It also means that the impact of infrastructural changes on regular development is mitigated, and improvements or changes to the environments can be made with lower risk. Currently, the decision to move to a new server room with different hardware would not pose a significant threat to the progress of feature development.

> *"The services were moved from their own small server room to [a third party]. Then we had to make a big jump. The operating system changed and everything else so we suggested that we rather do it so that we automate first and move then. Or maybe the migration was a pretext for automating everything related to the infrastructure and configurations, because we knew [...], if it's not automated at all, how much manual labor we would have to do."*
> — Developer 1

5.2 Customer benefits

Two customer stakeholders were interviewed for this study, and have presented their views on the benefits of CD adoption. The first stakeholder is an IT architect as part of the data administration team. The other is a lead user of one of the largest core applications under development by Solita, a role that can be compared to that of a product owner. Both have been part of the project at least since 2010.

5.2.1 Improved collaboration

On the customer side, too, the new ways of collaborating and working together were a central theme when discussing benefits from the improvements. One welcome aspect was that the responsibility of creating perfect specifications of changes no longer rests entirely on the shoulders of the lead user. Furthermore, the continuous communication and discussion regarding the specs and domain is appreciated.

> *"Solita are good in the way that they ask. If they don't know, they ask. It's not like they just make assumptions like 'could this possibly be like that', they pick up the phone or ask directly in Flowdock. That's a good way of doing things, because sometimes it can be like, we need something just because the law says so."*
> — Lead user

Even though this mode of working together can be seen as purely a prerequisite for CD, it has clearly also had an impact on the relationship between the two parties. Instead being purely a supplier, the customer appreciates the fact that the developers seem to care about them.

> *"There are suppliers with whom the customer experience is that they try to maximize billing during the agreement period [...].*

> *This service is more like a long-lasting companionship and not just short-sighted greed."* — IT Architect

The new communication tools, the shared Jira and Flowdock, are also contributing to the benefits of collaboration. The use of a shared Jira project has alleviated and made away with the issues of heavy bureaucracy that was present when having separate change management systems. Flowdock provides group discussions and archival of discussions, something which was not previously possible. Furthermore, the lower threshold to access the shared documentation in Confluence, enabled by simplifying the access and authentication process, means that developers are now more active on that platform. Naturally, the new tools would have little impact if developers were not constantly using them and if the change management process wouldn't have been revised.

> *"[The Confluence Wiki] is their daily tool, so if we reference something from there and tell them to comment it on Flowdock, they're already in. So the answer can come in 15 seconds, or a follow up question. Compared to the rigid ITIL and Change Management Board -style operating model we're quite lean now."* — IT Architect

5.2.2 Improved quality

There were definitive signs of product quality improvements based on the experiences of both developers and customer, but a considerable difference between the perceptions of the customer stakeholders must be noted. Quality improvement was clearly discerned by the more technically oriented of the two stakeholders. When outright asked about changes, the user perspective was that no distinguishable change in quality had occurred during the recent years. However, the quality assurance process was reportedly more complicated when several other suppliers were involved, leading to many errors in production. Furthermore, prototyping, which in essence intends to ensure a result more in line with the actual user requirements, was cherished by the lead user as well.

> *"Somehow I feel that [prototyping] is a much better way of doing things, than if we receive something that is done and I test it without even discovering [errors]. Then it's taken into use and change requests start coming, 'I'm missing this, and that has not been implemented'."* — Lead user

From the architect stakeholder perspective, many changes had contributed to the fact that the quality has increased. Considerably less bugs after production deployment is attributed to the adoption of automated testing, static code analysis, production-like environments in the deployment pipeline and major refactoring and platform improvement projects.

> *"Even though the main effort goes into features we have seriously seen the benefit of automated testing. [...] We don't see exceptions in production because the errors appear before that, and they are fixed. [...] When the worst flaws are removed and we operate by the principle that before release the [code analysis] trend line is negative [...], that is also seen as increased quality going towards the users into production."* — IT Architect

One reason for the disparity in perceived quality may be that many of the improvements have been made under the hood, in ways that are not obvious on the surface. The approach on quality seems to be to at least keep the quality of the code base from degenerating so as to achieve a certain level of maintainability. Larger undertakings regarding the platform, such as the migration from WebLogic to JBoss, were claimed to have extended the lifetime of some applications with several years. Furthermore, the user experience cannot be radically improved due to the legacy nature of the applications, which may be a reason for this phenomenon.

Another recognized quality attribute is defined by the deployment pipeline and the automated environment configuration: the potential speed of bringing changes to production. Developers now have the capability to deploy fixes much faster than before, even during the same day. The only current restriction is that deployments have to take place outside of office hours.

5.2.3 Increased productivity

The adoption of CD has enabled the customer stakeholders to be more productive. Prior to the changes detailed in this study, deployments could take from a few hours to the entire evening and night. Customer staff had to take care of preparation, send out notifications on the expected downtime to stakeholders and users, configure redirection of incoming requests and monitor the entire process until a stable deployment was confirmed. The situation now is considerably improved, and most of these tasks have been automated.

> *"With the current Ansible installation solution, the downtimes are between a few minutes and half an hour. If we go back four or*

> *five years the installer had an evening long project making sure the applications worked the next morning if there were no surprises. They had the source code with them and the surprices were fixed in the code during the evening after the first build failed."* — IT Architect

Streamlining the specification work and making requirements elicitation more agile and continuous has had its own impact on productivity in the customer organization. No longer are all features specified to the same extent regardless of the final solution to the requirements. The prototyping process is also part of this beneficial outcome, as making changes to prototypes requires less effort than writing up new change requests for features that are already in production.

> *"I like that I don't have to provide ready specifications, but that we can continuously specify along the way. [...] Before, when we were using Scrum, I think our specifications had to be pretty much finished. It's also been our internal way of working, that we wanted to specify really far the processes and procedures."* — Lead user

Another source of increased efficiency has been the improvements to the user acceptance testing. The coordination of testing has allowed the customer to focus more on interesting edge cases rather than testing everything. Furthermore, the Testiapina tool developed by Solita has reportedly cut down on acceptance testing effort for the users significantly.

> *"Yes [I think having more automated testing is a good thing]. Even the adoption of Testiapina has helped immensely since I don't need to prepare a lot. You can imagine how many fields have to be filled in when we process a project, how many estimates that need to be entered, all the classifications, the classifications based on the law and so forth. All these need to be filled in and it takes an ungodly amount of time. Now I get it with the click of a button, it copies the information from some other diary, so it has really sped up testing a lot. "* — Lead user

5.3 Summary

Both the developing organization and the customer stakeholders have perceived clear benefits from the adoption of CD practices. Most of the benefits

were identified to target the developers, including higher productivity, better ways of working together, lower risk of release failures and stress from deployments, and an over all better working environment with less menial tasks. Some benefits were identified by developers as being directly beneficial for the customer, such as flexibility of making infrastructural changes or increasing team size, and reduced risk of lost tacit knowledge. Customer stakeholders definitely recognized benefits too. The way tighter collaboration improved specification processes and mutual trust is complemented by increased product quality and more efficient use of time. While stakeholders' recognition of these benefits certainly tells us something about the results of the transformation, the next chapter investigates whether or not any improvements are actually measurable based on collected metadata.

Chapter 6

Measuring continuous delivery

As detailed in chapters 4 and 5, many changes and improvements have been made over the course of the last five years. New collaborative tools have been taken into use, the processes and methods have been refined to support a more agile and continuous way of working and the developers have started to actively communicate with customer stakeholders. In an effort to validate the perceived positive results of this transition, we collected historical metadata from development tools for analysis. In this chapter, the results of issue data analysis from the Jira tool is presented.

Analysis of the Jira issue data was complicated by the fact that the usage behavior and directions for use had changed several times over the years. As the processes and practices changed, so did the way issues were specified, recorded and managed. Furthermore, the account was previously split over several Jira projects, but has since been concentrated to one shared project. These challenges were mitigated by identifying attributes that were persistent, combining equivalent attributes and merging issues from all projects into a single dataset.

An issue in the Jira project can be thought of as a task or requirement from the customer. An issue has a set lifecycle based on different, pre-defined states. The states used in this project are detailed below. Further relevant issue attributes are labels and the version ID. The version ID can be used to identify when an issue has been deployed, using the timestamp of the version ticket's release.

Open
: The issue has been created by a stakeholder and a preliminary description of the task exists.

In progress
: A decision to start development of the issue has been made and it has

been assigned to a developer.

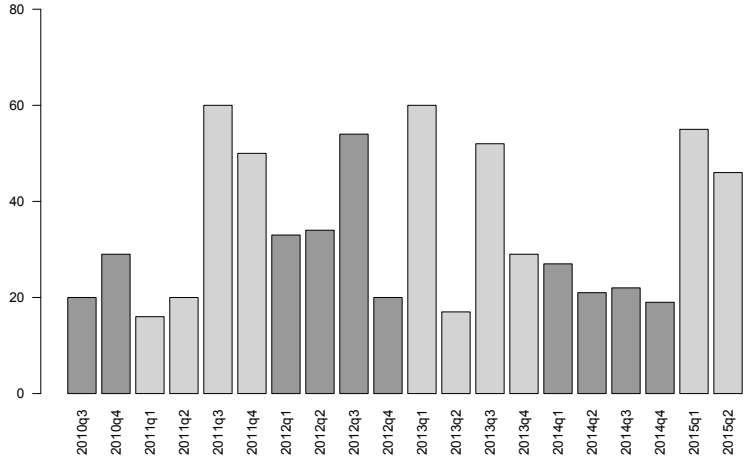
Resolved
Development of the issue is considered done by the developer and the changes have been reviewed by a peer.

Review
A customer stakeholder, commonly a lead user, performs acceptance testing of the changes made.

Closed
The changes have passed acceptance testing and the issue awaits deployment.

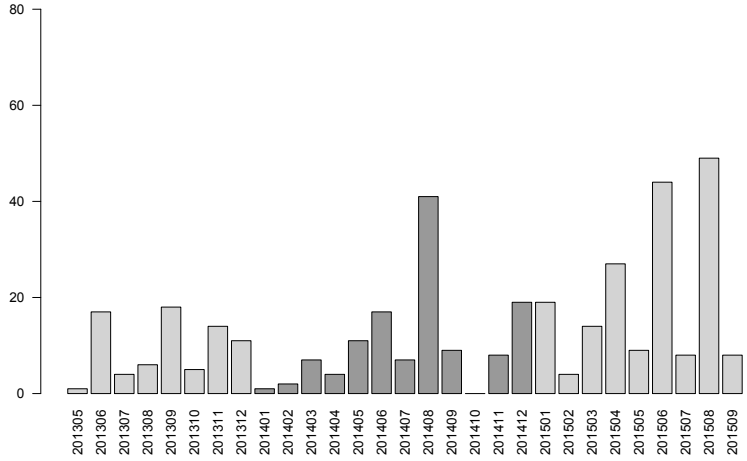
The timestamps of state changes allow us to calculate durations for different phases of the process. A documented benefit of CD is the capability of accelerated value delivery. The potential for faster deployments was also mentioned by the interviewees as a benefit of the transformation. By measuring the lead time for issues over the last five years, we can evaluate this proposed benefit. A diagram of the median lead time for issues resolved within a certain quarter can be seen in Figure 6.1. Lead time was calculated as the difference in time between setting the "In progress" state and the time that the issue is deployed into production.

Based on Figure 6.1 it seems evident that no clear trend of shorter lead times can be observed over the past five or so years. Instead, the median lead times fluctuate heavily between around 20 days and two months. These fluctuations are caused by the irregularity of projects (issue entities), their size and their tendency to block other development efforts. For example architectural improvement projects can lead to large delays in other development efforts that cannot be finished until the architectural changes are done. As mentioned in chapter 4, the development process was changed in 2013 to accommodate a Kanban lane specifically for the purpose of accelerating the delivery of small development items. These issues can be identified using at least two issue labels, "production bug" and "small development". A diagram of the median lead time for exclusively small development issues can be seen in Figure 6.2. Here, a clear difference is distinguishable. Small development efforts and urgently needed fixes that previously were part of larger projects can since 2013 usually be deployed within 10 days. Thus, while the average lead time for any issue has not clearly declined, there is a definitive improvement in the capability by which changes can be deployed when needed.



	2010	2011	2012	2013	2014	2015
Median	18	31	36	29	21	45
Mean	28	42	47	43	31	74
SD	62	40	51	49	41	77

Figure 6.1: Graph of median lead time of all issues resolved per quarter and summary table, measured in days.



	2013	2014	2015
Median	6	7	16
Mean	17	12	30
SD	22	24	40

Figure 6.2: Graph of median lead time of small development issues resolved per month and summary table, measured in days.

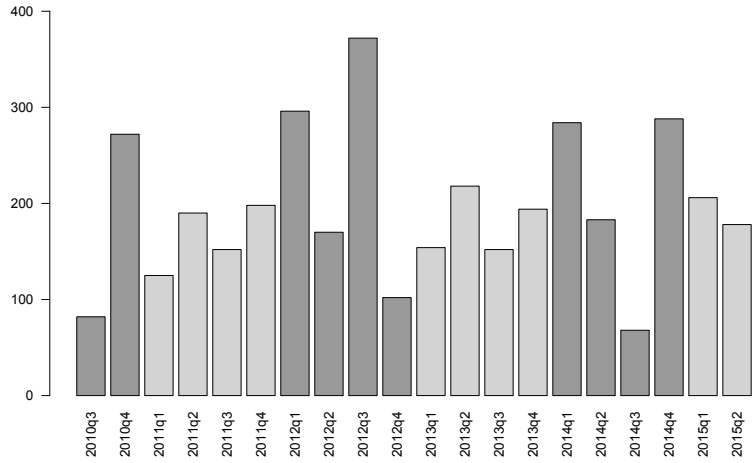


Figure 6.3: Number of issues resolved per quarter.

An obvious weakness of using Jira issues as a unit of work measurement is the fact that their size will undoubtedly be irregular. Developers mentioned that there are guidelines for approximately how much effort an issue should constitute, but that these are hardly followed or enforced. This fact is evident if we attempt measurement of development performance by issues resolved over time, as seen in Figure 6.3. There is no distinguishable trend in issues resolved per quarter. Thus, it is unlikely that the actual amount of work and effort over time is represented here, as both the amount of developers and applications under development have roughly tripled over the same period.

CD should also enable a higher release frequency according to both literature and stakeholders in this case. While related to the lead time of requirements, release frequency can also depend on the amount of applications under development, but does give some indication of how capable the developing organization are of releasing changes. A diagram of the number of releases per quarter can be found in Figure 6.4. Here, we have to note that a release does not necessarily indicate a single application installation, as application installations are often bundled up and deployed together. Over the last two years, individual installations, or singe application deployments, have been sitting quite steady at around 30 per month (according to production deployment logs collected since end of 2013).

Through smaller deployments, CD promises to lower the risk of release failure. By associating Jira issues with releases, it is possible to estimate the size of a release in terms of numbers of issues included. This solution is not

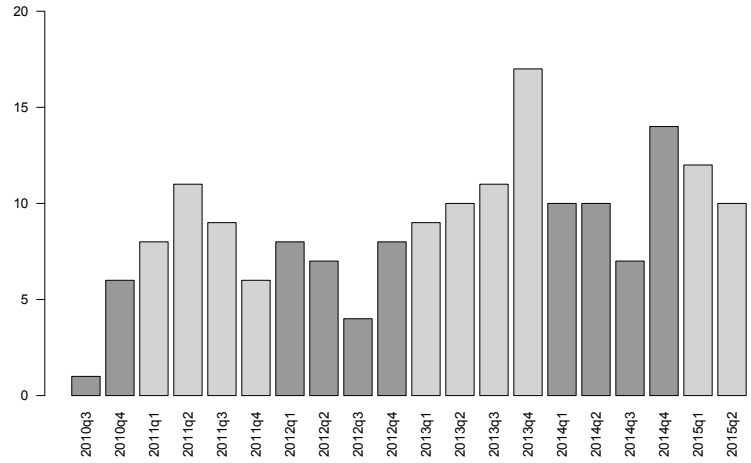


Figure 6.4: Number of application releases per quarter.

perfect, as there is no way of knowing the size of code changes that an issue required or the impact of those changes. However, it can give some indication of how many change entities are bundled together into a single release. In Figure 6.5, the amount released issues are visualized in a cumulative step diagram. Each release is seen as a rise on the y-axis, and the height of that rise shows the amount of issues included in the release. Here, we can distinguish a pattern. The large steps followed by plateaus present especially in 2013, indicate large releases followed by bug fixes. The most recent trend seems to be smaller, more frequent steps, indicating consistently sized releases and continuous development.

Code quality is another proposed benefit of CD that can be investigated through data analysis. The closest approximation of code quality evolution that could be attained through the data available is the amount of bugs that are opened. In Figure 6.6, the amount of bug reports made per quarter is visualized. No significant trend is visible. It is important to keep in mind, however, that these numbers include all bugs including those reported in acceptance testing. As such, the analysis is not representative of the quality of production deployments.

Even though several of the visualizations do not show clear improvements, as is the case with Figures 6.1 and 6.6, it may be relevant to keep in mind the growth of the account over the same period of time. As one developer claimed, it may not have been feasible to grow the account with the old processes and practices. Being able to keep these metrics from deteriorating significantly, while taking on triple the amount of developers and applica-

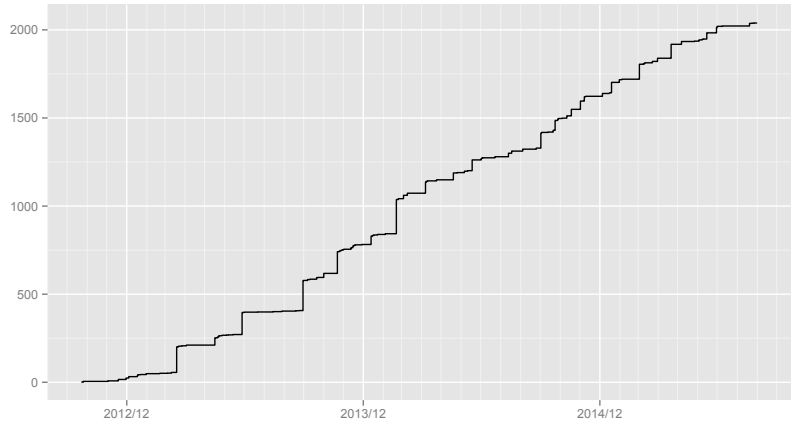


Figure 6.5: Cumulative number of issues released.

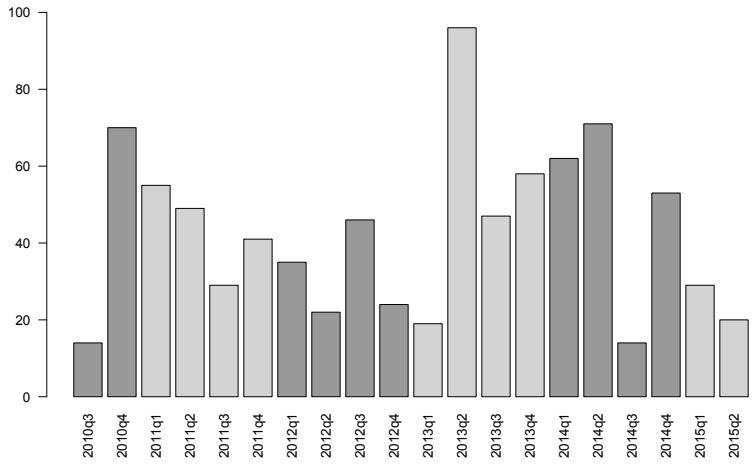


Figure 6.6: Total number of bug reports per quarter.

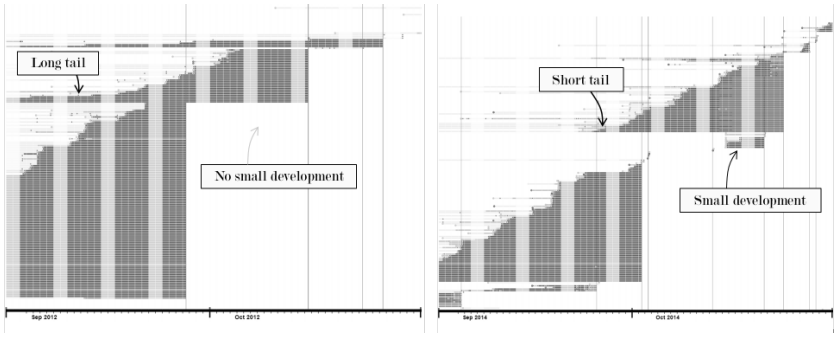


Figure 6.7: A comparison of Jira issue progress in late 2012 and 2014.

tions can be considered a feat in itself.

By portraying issues on a timeline and bundling them together according to releases, we can attempt to visualize the development mode described in Figure 4.1. In Figure 6.7, two excerpts of this type of visualization can be seen, a comparison between development in the late months of 2012 and 2014 correspondingly. While it is still difficult to draw conclusions on the performance of CD from this rough representation of development effort, the geometrics of it give an easy overview of comparable metrics. The height of one polygon represents release size, and is supposed to be kept low according to CD ideals. The length of the polygon, corresponding to the oldest issue part of the release, tells us how long the release has been worked on. In this particular comparison we can easily distinguish the lack of small development releases in 2012 and the longer tail of the last release in the same timespan, indicating that some issues lie waiting for deployment long after development is done. Furthermore, by the end of 2012, a release was made that is arguably too large according the ideal CD practices described in existing research.

Chapter 7

Discussion

7.1 Reflection

Research in the field of continuous delivery is scarce and very recent. Thus, the validity of prior work has not been proven through years of scrutiny, and the opportunities for reflecting the results of this study against existing literature are limited. This study aims to both validate and build on the small existing body of knowledge through comparable and new, previously undocumented findings.

7.1.1 RQ1: Adoption of CD

Regarding the process of adopting CD, the construction of a pipeline and the adoption of tools and practices, the case organization have achieved a situation strikingly similar to that described by many relevant authors. The deployment pipeline executes all the proposed steps presented in Section 2.2.1, albeit with room for some improvements in level of automation and testing methods and coverage. When it comes to the practical characteristics discussed in Section 2.2.3, the case scores close to full points. Automated testing was considered fast enough by the developers when omitting UI tests. The focus on deployable software was recognized by the feelings of relief over the new default green build status and green acceptance test and production server status. Organizational support clearly existed, although with some room for improvement on the customer side, and the collaboration was very much improved from the situation five years ago. Lastly, binaries are now only built once, and are independent of the environment they eventually end up in. If the organization wants to climb even closer towards the proposed ideal CD mode, it is advisable to focus on decreasing the size of code changes per release [Fowler, 2013] and increase the level of metric collection and

radiation [Humble and Farley, 2010]. Smaller changes could provide quicker value delivery, which would likely resonate well with the users that so far have limited insight into the benefits of all these changes. Richer metrics would provide a healthy understanding of the developers' performance, the impact of new tools and practices, and could be used to reason around future actions. Also, moving closer towards actual push button deployments by lowering the need for negotiating deployment dates would further pave way for more continuous practices.

If we disregard the literature and focus on the case from a contextual standpoint, there is little to criticize about what the stakeholders have achieved. The environment, the characteristics of the software and the surrounding organization and domain all pose significant threats to this type of undertaking. Automating anything for legacy code bases that have never been designed for automation is a challenge that is further accentuated by the high coupling and dependencies between applications. The public sector context puts restrictions on the ways of solving needs through laws and regulation, and stiff organizations reluctant to change are commonplace. One important enabler of the transition in this case was the freedom given to both developers and customer representatives by their corresponding organizations to do what they believe to be the best decisions in the long run. This allowed them to focus not only on new feature development. Another enabler is the fact that the customer organization had so called champions that understood the potential benefits of making changes to the way software is developed and could promote that mindset internally.

7.1.2 RQ2: Benefits of CD

Looking at the benefits resulting from the transition, some accomplishments can be recognized in all the themes identified in Section 2.3. Regarding the acceleration of value delivery, the potential has clearly improved, but the benefit is not obvious to all stakeholders. This aspect is particularly interesting as faster value delivery poses as one of the primary drivers for CD when looking at existing literature. This can be argued to show that there are other reasons for adopting CD that may weigh more heavily in certain contexts, as there was no evidence of it being a major target in the studied case. Deployments are clearly not as risky as they used to be and require less effort. Productivity has reportedly improved for almost all interviewees, as the customer spends less time on acceptance testing and redundant specification, and the developers spend less time on non value adding tasks. User feedback can technically be gained more quickly, but is limited by the schedules of the customer stakeholders. However, from developers to customer, the feedback

cycle showed great improvement. Both parties commented on the positive effects on software quality that the transition had provided, but here too it was least visible to the user. Arguably, many of the hazards mitigated by the changes, such as the risk of deployment failures, may not have been very visible to the users previously anyway.

However, the one benefit that outshone all others according to the interviewees was the collaboration. This is somewhat surprising, as the collaboration practices normally would be seen as an enabler or prerequisite for CD [e.g. Leppänen et al., 2015]. In this case, collaboration was clearly regarded as a reward in and of itself. Two other distinct benefits that had not been documented in the reviewed literature were the organizational and infrastructural agnosticism. Stakeholders from both sides of the account saw the ability to switch out both hardware and people without threatening the performance of the project as very valuable. These last two benefits are particularly interesting as they are somewhat unexpected perks and not normally a reason for pursuing CD. In this case, infrastructural agnosticism was in fact one of the major reasons for improving the environment independence and automation.

As an interesting side note, there were almost no negative perceptions of the changes. Even when asked neutrally about how they feel about the changes that have taken place, the interviewees mainly presented positive results. Only the lead user mentioned that, while the new ways of communicating are great, it can feel a bit exhausting to be available and active to the degree that collaborative development demands. Otherwise, the only directly negative experiences of the transition pertained to the challenging context, legacy code, monolithic applications and security and privacy requirements.

7.1.3 RQ3: Measuring continuous delivery

Through analysis of the Jira data it is indeed possible to measure some of the aspects of the software development process that CD intends to improve. However, the precision of the resulting metrics is only as high as the strictness of the tool usage. Regardless of the fidelity of the metrics, visualizations of the kind presented in the results can serve as a good basis for initial discussion and may capture some of the larger issues. For example, the large fluctuations in lead times may indicate difficulties in planning and knowing how long deployment of an issue will take. During the interviews, developers that were presented with graphics similar to those in chapter 6 were clearly enthusiastic about understanding them. One of the interviewees even began to dig through their own data in order to provide an explanation to an anomaly in the visualization. Metrics describing the results of e.g. a recent

tool implementation or process change can provide an objective perspective of phenomena that otherwise would be evaluated subjectively, and can engage stakeholders in discussion.

The type of visualization presented in Figure 6.7, where issues are grouped into releases and their state is visualized over time, has some potential for the evaluation of working practices. The dimensions of issues and projects reveal characteristics of the process, that can be enriched by qualitative insight. Some results of the CD adoption can be directly seen in this visualization, such as the introduction of small task development and deployment, and more findings and issues can probably be identified with additional experience. This could also prove to be a valuable tool for project management and stakeholder communication, providing a real-time view of the current status of issues against recent and previous history.

7.1.4 Future opportunities

Bringing the benefits to the users, both internal users and their customers, has so far been neglected to some degree. The transition has been more focused on the technical aspects and improvement of the working environment of developers. With the existing capabilities, there are opportunities to display the results of the efforts that sometimes have blocked necessary feature development for the internal users. Showing what is possible would likely help with changing the mindsets of stakeholders. For example, the requirements or at least desires for speed of feature delivery would likely be higher if the potential delivery times were made more visible. The same goes for the trust in automated testing. While technically minded stakeholders considered the project to be on the verge of being able to deploy continuously, the user perspective was that legal requirements and security cannot be guaranteed by automatic tests. For several of the applications, a fully automatic pipeline and continuous deployment is not an inconceivable possibility in the future. However, when it comes to the larger and older systems, rewriting them as modular and functionally decoupled solutions from the bottom up would likely be more profitable.

During the study it became evident that no modern usage analytics were employed. Having access to data on user behavior and being able to evaluate the functionality and value of existing and new features could bring along a clearer need for continuous delivery and deployment. Currently, the insight on how the applications are actually used is based largely on speculation and partly on user feedback through traditional challenge. One way that CD could bring value is by enabling continuous experimentation with new features. An incredibly short feedback loop could be established

if features could be deployed the instant they are done, and analytics be immediately collected. Iterating and experimenting with different potential solutions could quickly bring along a version that is more valuable than the one originally intended.

7.2 Threats to validity

Robert K. Yin (1994) makes the case that the quality of research design in a case study can be evaluated from four perspectives, or tests. The first, construct validity, is dealing with how well the selected methods capture and measure what the case study intended to examine in the first place. Secondly, internal validity is the measure of how strong the proof behind claims of causality between different phenomena in the study are. External validity, on the hand, is about the degree to which the findings can be generalized and the definition of the domain where the results are relevant. Last, reliability is the extent to which the same methods can be used to gain consistent results. [Yin, 1994]

In this thesis, the case study is largely descriptive in the qualitative parts and to some degree explorative in the quantitative section on measurements. No causal relationships have been argued as facts, since the research questions do not require it and there were few opportunities to triangulate the results. Due to the nature of this case study, internal validity is not to be evaluated [Yin, 1994]. There are parts of this thesis where a potential cause for a phenomena is suggested. The reader is to bear in mind that any causal relationships are speculations.

7.2.1 Construct validity

There are no direct flaws in the correlation between the research questions and the results. This study captured the type of phenomena that intended to examine. However, there are a few threats to construct validity that should be noted when interpreting the results. First, the low number of interviews that were held, considering the diversity of stakeholders, implies that the results aren't saturated. The entire case concerns stakeholders ranging from managers to developers in the developing organization; and customers, users and IT employees in the customer organization. What this means in terms of validity is that some of the results may be circumstantial, and all perspectives on the transition are unlikely to be covered. In some cases, results could not be triangulated, but this has been explicitly mentioned in the results by indicating a single source. Furthermore, it is conceivable that the interview

questions did not cover the entirety of the research topic. The follow-up questions and free form discussions in each interview helped with this issue, but these are not standardized methods yielding directly comparable results.

What concerns the quantitative analysis, a set of threats to construct validity exists: inconsistent issue size, validity of timestamps and inconsistent methods of use. It is reasonable to assume that the size of Jira issues varies, which in turn means that they are not directly comparable. There is, for example, no way of discerning the reasons behind the lifespan of an issue, as it is impacted by task size and complexity, wait times, dedicated developer effort etc. Furthermore, there is no way of validating that the timestamps of the state changes actually correspond to real-world state changes. For example, it was evident that stakeholders sometimes forget to change states, and usage isn't very strict. Lastly, stakeholders are likely to use any tool in different ways, according to their own habits, and Jira is probably no exception. In order to improve the validity of a quantitative analysis of this kind, additional sources should be used to cross reference the data. For example, VCS and CI server data can be used to provide some degree of triangulation if the commits and CI jobs can be linked to individual tasks and versions.

7.2.2 External validity

As this is a single case study, focussing on a very specific domain and context, one should approach any sort of generalization of the results with ample caution. Much of the complexity in the adoption of CD in this case stems from the challenges introduced by things like the application history and the domain. Thus, many of the choices made along the way have been selected and tailored to tackle these particular challenges. With adequate care, results can probably be somewhat generalizable to cases very similar in nature. Furthermore, results like the benefits of organizational and infrastructural agnosticism add to the existing body of knowledge on CD outcomes.

Regarding the measurement of CD, as stated in the discussion on construct validity, tools are used in different ways in different organizations. For example duration measurements, such as issue lead times, will only be as valid as the underlying practices of using the tool are. If there is no way of validating that the data corresponds with reality, the measurements should not be trusted. However, the metrics can still serve as a basis for discussion and for identification of anomalies.

7.2.3 Reliability

While most of the main results can plausibly be repeatedly deduced by using the methodology of this study, many of the findings are circumstantial in the sense that they were not explicitly investigated. The largest challenge regarding the reliability of the results is designing an interview that yields consistent results. This requires further iterative improvement of the question set and a larger pool of interviewees than feasible in this case. As mentioned in the evaluation of construct validity, some findings stem from the fact that one or several interviewees considered them important in the context of the study, rather than having been explicitly asked about them. Thus, without a more thorough approach to interviews, both regarding interview size and amount of interviewees, the qualitative results are not repeatable.

The methods used for quantitative data analysis are reliable in the sense that they will produce the same results when repeated. However, it is important not to draw assumptions directly from those results, without understanding the implications on them. The metrics produced through the analysis are only reliable and bear meaning if the underlying process, e.g. the practice of issue state changes, is clearly understood.

7.3 Future research

Several aspects of continuous delivery arose as potential themes for future research. One such theme is the needs for CD. While we have a documented set of benefits and some pointers on how to achieve those benefits, nothing has previously been written about the needs for CD, where they come from, what has caused them and what they mean. For example, in this case the outspoken need for CD from the actual users, those for which the entire development effort is intended, was surprisingly vague. While the developers wanted to be able to deliver changes in a day, a user was satisfied with delivery in a few weeks. As such, the undertaking wasn't entirely user value driven, and it seemed that no significant effort had been made to understand the user perspective on development methods. This is relevant for organizations that want to pursue CD, as they somehow have to decide on an appropriate level of sophistication and try to target certain goals if the transition is to be cost effective. In this case, stakeholders agreed that all improvements may not have been feasible if not for the decent budget.

Another identified topic for future research is the measurement of CD and DevOps in general. There are multiple angles that make this theme relevant. First, as was discovered in this study, an organization needs to make

a conscious choice to collect data from many tools and systems. Historical metadata is almost irreplaceable when trying to evaluate if a change, such as the adoption of a new tool has been successful. Without setting goals for a change according to some metric, and gathering and analyzing the data necessary to evaluate those goals, we have to resort to the opinions of stakeholders to validate the change. There is also plenty of room from the development of new ways of measuring CD, new metrics that capture the pursued benefits more accurately than the traditional metrics presented in this study. The final aspect of measurement that would be interesting to examine is how metrics can be used as a tool for communication with the customer and the users. Metrics could potentially be used to change mindsets, argue the needs for changes and, of course, to demonstrate results.

Chapter 8

Conclusions

According to the findings, the case organization has indeed managed to implement many changes that have brought the development mode all the way from the traditional waterfall model with no automation to speak of into the modern world of assistive tools and collaborative value driven development. The changes have taken place over the course of around five years. The major enablers of this evolution have been allowing individuals to make improvements according to their needs, removal of any obstacles and thresholds for continuous communication, and a supportive customer organization. The transition has not been without its challenges. The monolithic, heavily coupled legacy code base proved to impair the adoption of automation, both in regards to configuration, testing and deployment. The fact that small changes impact many applications limits how small the code changes can be for a deployment and often artificially enlarges the release sizes. The public sector domain limits the range of possible solutions and raises the requirements on security and privacy, which in turn impedes developer access and lowers the ratio of automatic to manual testing. This study shows that CD is achievable despite this difficult context. Not all solutions at the time of writing are optimal according to the CD ideals, there is no true push-button deployment practice and the pipeline contains more than one manually triggered stage. Despite this, the changes have provided perceived benefits for all stakeholders involved in the study. When asked, there was not a single suggestion that the changes had not been for the better. Improved productivity, software quality and work life are just a few of the improvements that were highly regarded by the stakeholders. Furthermore, this study identified benefits that have not previously been documented, infrastructural and organizational agnosticism. When comparing the benefits of different parties, it is obvious that the changes impact the work of different roles in different ways, but also that the major benefits are perceived by the developers and

technically minded stakeholders. Recommended improvements for the developing organization is to focus on the user perspective and needs, and to specify, gather and analyze the data needed for objective validation of the success of changes and tool adoptions.

As this is a single case study, the results cannot be widely generalized. However, the study can provide valuable insight into what is potentially achievable in a similar context. The challenges that are detailed serve to acknowledge areas that may need extra attention in a comparable domain. Furthermore, the importance of understanding the ways of working and the ways tools are used has shown to be important in the selection and interpretation of software process metrics. For future research on the measurement CD, it is recommendable to identify such metrics that are principally affected by the changes they are intended to validate, and that stem directly from user needs.

Bibliography

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000. ISBN 0201616416.

S. Bellomo, N. Ernst, R. Nord, and R. Kazman. Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 702–707, June 2014. doi: 10.1109/DSN.2014.104.

V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi: 10.1191/1478088706qp063oa.

L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015. doi: 10.1109/MS.2015.27.

G.G. Claps, R. Berntsson Svensson, and A. Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57(0):21 – 31, 2015. doi: http://dx.doi.org/10.1016/j.infsof.2014.07.009.

M. Fowler. Continuous Integration, May 2006. URL `http://martinfowler.com/articles/continuousIntegration.html`.

M. Fowler. Continuous Delivery, May 2013. URL `http://martinfowler.com/bliki/ContinuousDelivery.html`.

J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, Upper Saddle River, NJ, 1 edition edition, August 2010. ISBN 9780321601919.

M. Leppänen, S. Mäkinen, M. Pagels, V-P. Eloranta, J. Itkonen, M.V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, 2015. doi: 10.1109/MS.2015.50.

Steve Neely and Steve Stolt. Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). pages 121–128. IEEE, August 2013. doi: 10.1109/AGILE.2013.17.

H.H. Olsson, H. Alahyari, and J. Bosch. Climbing the "Stairway to Heaven"; – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 392–399, September 2012. doi: 10.1109/SEAA.2012.54.

M.Q. Patton. *Qualitative Research & Evaluation Methods*. SAGE Publications, 3rd edition, January 2002. ISBN 0761919716. Published: Hardcover.

A. Strauss and J.M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998. ISBN 9780803959408.

D. Ståhl and J. Bosch. Automated Software Integration Flows in Industry: A Multiple-case Study. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 54–63, New York, NY, USA, 2014a. ACM. doi: 10.1145/2591062.2591186.

D. Ståhl and J. Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87: 48–59, January 2014b. doi: 10.1016/j.jss.2013.08.032.

R.K. Yin. *Case study research: Design and methods*, volume 5. SAGE Publications, second edition edition, 1994. ISBN 978-0803956636.